IBM FORTRAN/2™ Computer Language Series

Fundamentals

Programming Family

IBM

84X1752

IBM FORTRAN/2™ Computer Language Series

Fundamentals

Programming Family

IBM

First Edition (September, 1987)

The following paragraph does not apply to the United Kingdom or any country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This publication could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time.

It is possible that this publication may contain reference to, or information about, IBM products (machines and programs), programming, or services that are not announced in your country. Such references or information must not be construed to mean that IBM intends to announce such IBM products, programming, or services in your country.

Requests for copies of this publication and for technical information about IBM Personal Computer products should be made to your authorized IBM Personal Computer dealer or your IBM Marketing Representative.

Preface

Library Guide

The IBM FORTRAN/2 library consists of three manuals. The following chart shows where different information is located.

If You Want to...	Refer to...
Install the product	Compile, Link, and Run
Learn basic facts about the language	Fundamentals
Know the syntax of an instruction	Language Reference
Understand error messages	Language Reference
Debug a program	Compile, Link, and Run
Compile a program	Compile, Link, and Run
Link a program	Compile, Link, and Run
Write a program	Fundamentals, Language Reference, and Compile, Link, and Run

Inside This Book

This manual introduces IBM FORTRAN/2, a FORTRAN language that can be used with the IBM Math Co-Processor. By increasing the speed of all numeric and trigonometric functions, the math coprocessor makes IBM FORTRAN/2 ideal for scientific and technical use. Although written to take advantage of the math coprocessor, IBM FORTRAN/2 provides a compiler option that emulates the functions of the coprocessor.

IBM FORTRAN/2 is designed according to the specifications of the following industry standards as understood and interpreted by IBM as of September, 1987:

- American National Programming Language FORTRAN 77 (ANSI X3.9-1978)
- The Code for Information Exchange (ANSI X3.41-1977)
- The Code Extension Techniques for use with the 7-bit Coded Character Set of American National Standard for Information Exchange (ANSI 3.41-1974)
- The IEEE Standard 754 for floating point arithmetic, with the following differences:
  - Rounds to nearest mode only
  - Rounding precision mode
  - No trapping (signaling) NaNs (not a number)
  - No user traps
  - Exception status flags are not supported

In addition, IBM FORTRAN/2 contains many useful extensions to that language beyond the ANSI X3.9-1978 standard. This manual explains those extensions as well. See Chapter 4, "Portability, Conversion, and Extensions" in the *IBM FORTRAN/2 Language Reference* manual for more information about the extensions.

Related Publications

The following books contain topics related to information in the IBM FORTRAN/2 library.

- *IBM Disk Operating System*
- *IBM Disk Operating System User's Guide*
- *IBM Disk Operating System Technical Reference*

- *IBM Operating System/2™ User's Guide*
- *IBM Operating System/2 User's Reference*
- *IBM Operating System/2 Programmer's Guide*
- *IBM Operating System/2 Technical Reference*

- *IBM Personal Computer Guide to Operations*
- *IBM Personal Computer Technical Reference*

- *IBM Personal System/2 Model 50*
- *IBM Personal System/2 Model 60 Technical Reference*
- *IBM Personal System/2 Model 80 Technical Reference*

Audience Statement

This manual is designed for people who have programmed previously and who are familiar with IBM Disk Operating System (DOS) or IBM Operating System/2 (OS/2).™[1] If you are not familiar with either of these operating systems, refer to the operating system manuals for information about them.

---

1 Operating System/2 and OS/2 are trademarks of the International Business Machines Corporation.

Contents

Chapter 1. Introduction

About This Book

This manual is a part of the IBM FORTRAN/2™[1] documentation set. Its purpose is to introduce you to the IBM FORTRAN/2 language. The manual is organized into the following chapters:

- Chapter 1, "Introduction" describes the purpose of the manual and identifies the files and resources you need to successfully write, compile, link, run, and debug programs on your IBM Personal Computer using the IBM FORTRAN/2 compiler.
- Chapter 2, "General Information" describes IBM FORTRAN/2 data types, names, expressions, and coding conventions.
- Chapter 3, "Program Structure" describes the relationship between the main program unit and all forms of subprograms.
- Chapter 4, "File Processing" describes the IBM FORTRAN/2 file system and provides information about input and output.

Note: This manual is not a tutorial; rather, each section explains one aspect of the IBM FORTRAN/2 language.

Additional Documentation

The other manuals of this documentation set are the *IBM FORTRAN/2 Language Reference* manual and the *IBM FORTRAN/2 Compile, Link, and Run* manual. They serve as a guide to the IBM FORTRAN/2 language, and explain how to compile, link, run, and debug programs written in IBM FORTRAN/2. These manuals are organized as follows:

[1] IBM FORTRAN/2 is a trademark of the International Business Machines Corporation.

IBM FORTRAN/2 Language Reference manual:

- Chapter 1, "Introduction" describes the purpose of the manual and identifies the files and resources you need to successfully write, compile, link, run, and debug FORTRAN programs on your IBM personal computer using the IBM FORTRAN/2 compiler.
- Chapter 2, "Statements" provides a complete description of the statements that are available for use with the IBM FORTRAN/2 language — including the purpose and correct coding for each statement.
- Chapter 3, "I/O Editing" is a guide for the use of the input/output statements presented in Chapter 2 — including the various editing commands available in the IBM FORTRAN/2 language.
- Chapter 4, "Portability, Conversion, and Extensions" describes the issues related to using the IBM FORTRAN/2 compiler to compile existing FORTRAN programs and to run programs compiled with the IBM FORTRAN/2 compiler on different computers.
- Appendix A, "Messages" lists the messages and codes for errors and warnings that are reported by the compiler, linker, runtime system, and debug facilities.
- Appendix B, "Floating Point Operations and Exceptional Values" contains tables and references that describe floating-point operations and explain how exceptional values (infinities and NaNs) are generated and written.
- Appendix C, "Code Optimization" describes code optimization.
- Appendix D, "Intrinsic Functions" contains a complete table of IBM FORTRAN/2 intrinsic functions and notes on using them.
- Appendix E, "Additional Routines" contains information about routines which may be helpful in using IBM FORTRAN/2.
- Appendix F, "Internal Data Representation" describes the internal representation of data.
- Appendix G, "Limits and Ranges" lists the limits and ranges of IBM FORTRAN/2 programs, I/O, and data.
- Appendix H, "ASCII Codes" lists all the ASCII codes (in decimal) and their associated characters.

- Appendix I, "Hollerith and Hexadecimal Data" contains instructions for using Hollerith and hexadecimal data types.

Note: A glossary of terms used with IBM FORTRAN/2 follows the appendixes.

IBM FORTRAN/2 Compile, Link, and Run manual:

- Chapter 1, "Introduction" describes the purpose of the manual and identifies the files and resources you need to successfully write, compile, link, run, and debug programs on your IBM personal computer using the IBM FORTRAN/2 compiler.
- Chapter 2, "Installation" describes how to install the IBM FORTRAN/2 compiler, create work files, and edit the configuration file. The chapter also explains the procedures for compiling, linking, and running a sample IBM FORTRAN/2 program.
- Chapter 3, "Compiling Your Program" describes in more detail how to compile the contents of a source file to create an object module, obtain a compiler listing (including error messages), and redirect a compiler listing to a desired disk file.
- Chapter 4, "LINK: The Object Linker" describes the actions performed by the linker and identifies the IBM FORTRAN/2 Libraries that can be called by an object module.
- Chapter 5, "Running Your Program" describes how to start and cancel execution of a program under DOS and OS/2, use IBM FORTRAN/2 runtime routines, react to runtime errors, and control various I/O devices during runtime.
- Chapter 6, "LIB: The Library Manager" describes how to use the library manager to store and retrieve object modules.
- Chapter 7, "Debugging Your Program" describes how to set up your configuration to accept Debug commands and display Debug output, stop a program and enter a Debug command, and efficiently use Debug commands and Debug device defaults.
- Chapter 8, "Interfaces with Other IBM Languages and Products" describes how to define subprograms written in Assembler language.

What You Need

To successfully write, compile, link, run, and debug programs on your IBM personal computer using the IBM FORTRAN/2 compiler, you need:

- IBM FORTRAN/2 master diskettes.
- IBM Personal Computer Disk Operating System (DOS), Version 3.30; or IBM Operating System/2 (OS/2), Version 1.0.
- One of the following systems:
  - IBM Personal Computer
  - IBM Personal Computer XT™[2]
  - IBM Personal Computer AT® [3]
  - IBM Personal Computer PC Convertible
  - IBM Personal System/2™ Models 30, 50, 60, or 80.

  With DOS, a minimum of 320 KB available user memory is required. With OS/2, a minimum of 1.5MB available user memory is needed.
- Monitor with 80-column capacity.
- A math coprocessor compatible with your system (optional, but recommended).
- A fixed disk and one 1.2MB, 1.44MB, 720KB, or 360KB diskette drive; *or* two diskette drives, each of which can be 1.2MB, 1.44MB, 720KB, or 360KB. A fixed disk is recommended.
- A printer (optional, but recommended).

---

[2] Personal Computer XT and Personal System/2 are trademarks of the International Business Machines Corporation.

[3] Personal Computer AT is a registered trademark of International Business Machines Corporation.

What You Have

IBM FORTRAN/2 Software

IBM FORTRAN/2 software is provided on four 360KB (5¼") diskettes and two 720KB (3½") diskettes. These are referred to as the master diskettes. The 360KB diskettes are:

IBM FORTRAN/2 "INSTALL" Master Diskette (360KB)

Filename	Contents
README	A list of changes to IBM FORTRAN/2 not included in the manuals
PACKING.LST	Packing list of delivered software
INSTALL.EXE	Installation program for DOS
FORTRAN.EXE	IBM FORTRAN/2 Compiler
FORTRAN.CER	Compiler Error Messages
DEMO.FOR	Demonstration Program
DEMO.DCM	Demonstration Program Debug Command Input File
QFORT.BAT	Sample Compiler Batch File
QFORTLNK.BAT	Sample Link Batch File
QFORTRUN.BAT	Sample Runtime Batch File

IBM FORTRAN/2 "LINK_RUN" Master Diskette (360KB)

Filename	Contents
FORTRAN.ERR	Runtime Messages
FORTRAN.DER	Debug Messages
FORTDBG.HLP	Debug online help facility
FORTRUN.DLL	Dynamic runtime and debug
FORTRUE.DLL	Dynamic runtime, debug, and emulator
LINK.EXE	Linker
LINK.PIF	Linker Program Information File for TopView
LIB.PIF	Library Manager Program Information File for TopView
FORTILC.ASM	Assembly source for inter-language calls
COPYMEM.ASM	Assembly source for copying memory
FORTRAN.PIF	Program Information File for TopView

IBM FORTRAN/2 "NP_LIBRARY" Master Diskette (360KB)

Filename	Contents
FORTRRN.LIB	Runtime and Debug library
FORTRIN.LIB	Intrinsic function library
FORTRDN.LIB	Runtime and Debug import library
PCDOS.LIB	OS/2 to DOS mappings library
LIB.EXE	Library Manager
FORTRAN.PIP	Program Information Profile for OS/2
FORTPIP.LIB	Program Information Profile library

IBM FORTRAN/2 "EM_LIBRARY" Master Diskette (360KB)

Filename	Contents
FORTRRE.LIB	Runtime, Debug, and emulator library
FORTRIE.LIB	Intrinsic function library
FORTRDE.LIB	Runtime, Debug, and emulator import library
PCDOS.LIB	OS/2 to DOS mappings library
LIB.EXE	Library Manager

The 720KB (3½") diskettes provided with IBM FORTRAN/2 are:

IBM FORTRAN/2 "INSTALL" Master Diskette (720KB)

Includes the contents of the 360KB IBM FORTRAN/2 "INSTALL" and IBM FORTRAN/2 "LINK_RUN" master diskettes.

IBM FORTRAN/2 "LIBRARY" Master Diskette (720KB)

Includes the contents of the 360KB IBM FORTRAN/2 "NP_LIBRARY" and IBM FORTRAN/2 "EM_LIBRARY" master diskettes.

Summary of Changes

Listed below are features of IBM FORTRAN/2 that differ from the predecessor product, IBM Professional FORTRAN.

- A single copy of the compiler that runs in all modes of OS/2 and DOS Version 3.30.
- Generated code that can be linked for all modes of OS/2 and DOS Version 3.30.
- Code segments of compiler and generated code that can be shared.
- Calls that can be made to the OS/2 Application Programming Interface (OS/2 only).
- Interactive Debugger with enhanced display capabilities and new commands.
- New compiler options and environment variables.
- Mainframe compatibility enhancements.
- Isolation of all textual messages to facilitate language translation.
- Support for file sharing over the IBM PC Network and in a multitasking environment on OS/2.
- Math Co-Processor emulation.
- Installation aids for various system configurations.
- Overlay support for DOS.
- IMPLICIT NONE statement.
- EJECT statement.
- Increased limits and ranges.
- A compiler option to create code specific to various system configurations.
- Threaded compiler error messages.
- Backslash edit control descriptor.
- Minus carriage control character.

- Comma external field terminator.
- OPEN with ACCESS = 'APPEND'.
- Alternate version of the SNGL function that forces rounding to single precision.
- New DSNGL function that forces rounding to double precision.
- Improved Assembly Language support (FORTILC).
- Alternate character length function (CLEN).
- Additional compiler listing options.
- Compiler status messages.
- COMPLEX*16.
- Mixing of complex and double precision values in the same expression.
- Additional code optimization.
- Underscore allowed as a non-leading character in a symbolic name.
- $ as high end of implicit range.

Chapter 2. General Information

This chapter provides general information about IBM FORTRAN/2. It describes:

- Coding conventions
- Character set
- Data
- Data types and constants
- Symbolic names
- Arrays
- Character substrings
- Storage allocation
- Expressions
- Hierarchy of expressions and operators
- Executable and nonexecutable statements
- Ordering of statements and lines.

Coding Conventions

Statements

An IBM FORTRAN/2 source program can be considered a sequence of statements. An IBM FORTRAN/2 statement consists of an initial line of source code and, optionally, continuation lines and comment lines.

The lines are divided into 80 columns. A character can appear between columns 1 through 80, inclusive.

The columns that are significant in IBM FORTRAN/2 are:

Columns	Contents
1 - 5	Statement labels and comment indicators. Comment indicators appear in column 1 only.
6	Continuation indicators.
7 - 72	Source statements.
73 - 80	Program identifier, which is not recognized by the compiler.

A tab character placed in columns 1-6 of a line causes the next character to be interpreted as being in column 7. The tab is expanded to the appropriate number of blanks in the listing file. All other tabs are passed as blanks to the listing file and are treated as blanks in character constants. The tab character has an ASCII value of hexadecimal 09 and is generated when you press the Tab key.

Note: The use of tabs in the source line is an extension to the ANSI X3.9-1978 FORTRAN 77 standard.

Lines

A line in an IBM FORTRAN/2 source program is either the initial line of a statement, a continuation line of a statement, or a comment line.

Initial Lines

An initial line is the first, or only, line of an IBM FORTRAN/2 statement. It contains either a blank, a 0, or a tab in column 6 and is not a comment line. The first five columns of the line must contain blanks or a label. Column 1 can contain a D to indicate a conditionally compiled statement.

Continuation Lines

A continuation line is a line that contains any character in column 6 other than a blank, a 0, or a tab. It is not a comment line. The first five columns of a continuation line must contain blanks. Continuation lines give you more lines in which to write a statement. There is no limit to the number of continuation lines in a statement, other than available memory.

Notes:

1. This is an extension to the ANSI X3.9-1978 FORTRAN 77 standard, which provides a maximum of 19 continuation lines. The ANSI X3.9-1978 FORTRAN 77 standard also specifies that the continuation line character must be from the ANSI X3.9-1978 FORTRAN 77 standard character set.
2. INCLUDE, EJECT, and END statements cannot be continued.

When continuation lines are used, a 0 can be included in the continuation column on the initial line to improve readability. This is illustrated in the following example:

Example

```
Column
1     567

C      AN EXAMPLE OF CONTINUATION LINES
      0IF ((A.LT.0).AND.(B.LT.0)
      1.AND.(C.LT.0))
      2THEN
        J = 2
        END IF
```

Comment Lines

A line is treated as a comment when it contains any of the following:

- C (or c) in column 1.
- * in column 1.
- All blanks.

Comment lines are useful for documentation and for separating sections of code to make the program more readable. They can appear anywhere in a program and do not affect its execution in any way.

The following example shows how comment lines can be coded:

Example

```
Column
1    567

C       THESE COMMENT LINES ARE FOLLOWED BY AN
C       INITIAL LINE AND A CONTINUATION LINE
     50 A=B+C
       *-D
```

Conditionally Compiled Statements

A line with the letter D (or d) in column 1 begins a conditionally compiled statement. When the /D compiler option is used during compilation, the line is compiled. Otherwise, the line is treated as a comment line.

Typically, the conditional compile feature is used to include debugging output lines during program checkout and to exclude such lines in the final object program.

Note: This feature is an extension to the ANSI X3.9-1978 FORTRAN 77 standard.

The following example shows how conditionally compiled statements can be coded. The example prints every input value for X when you compile using the /D compiler option.

Example

```
Column
1    567

       READ *,X
C      ECHO THE INPUT DATA IF DEBUG
D      PRINT *, 'INPUT DATA X =',X
       Y = X**3
```

For more information about compiler options, see Chapter 3, "Compiling Your Program" in the *IBM FORTRAN/2 Compile, Link, and Run* manual.

Labels

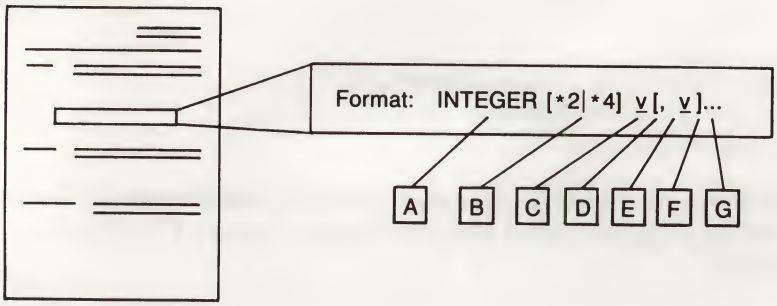
A label can be included in a statement. The statement label is a sequence of 1 to 5 digits that is unique in each program unit. At least one digit must not be 0.

A label can be placed anywhere in columns 1 through 5 of an initial line. Blanks and leading zeros are not significant in distinguishing between statement labels.

Any executable or nonexecutable statement can have a label. However, nonexecutable statements (except the FORMAT statement) cannot be referenced. See "Executable and Nonexecutable Statements" on page 2-46 for an explanation and listing of executable and nonexecutable statements.

Notation Conventions

The following example illustrates the notation conventions for IBM FORTRAN/2 statements.



A Uppercase words	are IBM FORTRAN/2 keywords and must be entered as shown.
B Vertical bar |	separates a choice of elements. Use only one of the elements shown.
C Lowercase letters	represent user-supplied data names and constants.
D Punctuation marks	such as commas and apostrophes are IBM FORTRAN/2 special characters and must be entered as shown.
E Blanks	have no meaning except where noted.
F Square brackets [ ]	enclose optional elements.
G Ellipsis ...	shows that an element can be repeated indefinitely.

Character Set

Any of the letters, digits, and special characters in the IBM FORTRAN/2 character set can be used to code program statements. Those letters, digits, and special characters are described below.

Letters: A through Z, a through z, and $

Digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9

Special Characters:

blank	) right parenthesis
= equals sign	, comma
+ plus sign	. decimal point
– minus sign	: colon
* asterisk	\ back slash
/ slash	' apostrophe (single right quote)
(left parenthesis	> greater than sign
< less than sign	_ underscore

Other ASCII characters can appear in an IBM FORTRAN/2 statement only as part of a character or Hollerith constant. See Appendix H, "ASCII Codes" in the *IBM FORTRAN/2 Language Reference* manual for more information about ASCII characters. Any printable character can appear in a comment.

Except for character constants and some format descriptors, no distinction is made between uppercase and lowercase letters.

Note: The use of $ as a letter, lowercase letters, and the <, >, _, and \ characters are extensions to the ANSI X3.9-1978 FORTRAN 77 standard.

The collating sequence for the IBM FORTRAN/2 character set is the ASCII sequence. See Appendix H, "ASCII Codes" in the *IBM FORTRAN/2 Language Reference* manual for a complete set of the ASCII character codes.

Blanks

Blanks have significance only in character and Hollerith constants, some format specifications, and column 6 of the source code. You can use blanks anywhere else within the program to make it more readable.

Blanks can also be embedded in IBM FORTRAN/2 keywords. For example, the following keywords are equivalent:

```
GOTO
GO TO
G O T O
```

Data

The data in a program unit is represented by constants, variables, arrays, substrings, and dummy arguments.

Constants

A *constant* is an explicitly stated value which does not change during program execution. For example, the integer seven is represented as "7"; the number pi, to four significant digits, is represented as "3.142". A constant can also be represented by a symbolic name. See the PARAMETER statement section of Chapter 2, "Statements," in the *IBM FORTRAN/2 Language Reference* manual for information about assigning a symbolic name to a constant.

Variables

A *variable* is an entity having both a name and a type, whose value can vary during program execution. A symbolic name represents a variable and identifies its storage location. DELTA is an example of a variable name.

Arrays

An *array* is an ordered set of data occupying consecutive storage. This set of data has a name and a type.

A symbolic name used to represent an array, for example, MATRIX, is declared in an *array declarator.* Each unit of data in an array is an *array element.*

An array element is represented by an array name qualified by a subscript list enclosed in parentheses. For example, MATRIX(2,6) is an array element name.

The number of elements in an array is declared along with the array name in the array declarator. The number of array elements is defined by a list of array dimensions. The value of an array element can change during program execution.

Note: Although an array element conforms to the definition of a variable, the term "variable" is not used in this manual to refer to an array element. See "Array Elements" on page 2-24 for more information.

Substrings

Character data is a variable or array element containing data of type character. The position of a particular character within character data is designated by a character position number.

A set of one or more consecutive character positions within character data is a *substring.* See "Data Types and Constants" on page 2-10 for more information about substrings.

Dummy Arguments

A *dummy argument* in a procedure has one of two forms: a symbolic name or an asterisk. A *symbolic name dummy argument* is a variable, array, or procedure that is associated with the actual argument when the procedure is referenced. An *asterisk dummy argument* indicates that the corresponding actual argument is an alternate return specifier. See Chapter 3, "Program Structure" for information on using dummy arrays, dummy procedures, and alternate return specifiers.

Data Types and Constants

IBM FORTRAN/2 has five basic data types:

- Integer
- Real
  - –Single-precision
  - –Double-precision
- Complex
  - –Single-precision
  - –Double-precision
- Logical
- Character.

This section describes the properties, range of values, form of constants, and storage requirements for each type.

Integer

The *integer* data type is a subset of the negative and positive numbers, and zero. An integer value is an exact representation of the corresponding integer, and occupies 2 or 4 bytes of storage. An integer can be specified in IBM FORTRAN/2 as INTEGER*2, INTEGER*4, or INTEGER.

INTEGER*2 specifies a 2-byte integer, while INTEGER*4 specifies a 4-byte integer. INTEGER specifies either 2-byte or 4-byte integers, depending on whether you use the /I compiler option. (The default is 4 bytes.) For more information about compiler options, see Chapter 3, "Compiling Your Program" in the *IBM FORTRAN/2 Compile, Link, and Run* manual.

Note: The /I compiler option violates the ANSI X3.9-1978 FORTRAN 77 standard rules for INTEGER data type allocations in COMMON and EQUIVALENCE statements.

A 2-byte integer can contain any value in the range −32,768 to 32,767. A 4-byte integer can contain any value in the range −2,147,483,648 to 2,147,483,647.

Integer constants are one or more decimal digits that are preceded by an optional arithmetic sign: plus (+) or minus (−). A decimal point is not allowed in an integer constant. The following are examples of integer constants:

```
   123          +123          -123     Ø
ØØØØØ123       32767        -32768
```

An integer constant has INTEGER type. Its size can be 2 bytes or 4 bytes depending on the /I compiler option. The default is 4 bytes.

Real

The *real* or *single-precision* data type (REAL or REAL*4) is written as an optionally signed string of decimal digits that includes a decimal point, an exponent, or both. The exponent follows the numeric value and consists of the letter E (or e) followed by an optionally signed integer constant. The integer represents the power of 10 by which the value is multiplied. A real data type is the same as a floating-point data type.

A real (single-precision) value occupies 4 bytes of storage. The precision is approximately 8 decimal digits.

For example, the following are all single-precision data types for the number 1230:

```
Ø.123E+Ø4      1.23e3            1230.
12300E-Ø1       123E1      1230.ØØØEØØØ
```

The range of real values is:

Smallest Value	Largest Value	
1.2*10**–38	3.4*10**38	Normalized
1.4*10**–45	1.2*10**–38	Denormalized

Note: See Appendix D, "Intrinsic Functions" in the *IBM FORTRAN/2 Language Reference* manual for an explanation of normalized and denormalized numbers.

Double Precision

The *double-precision* data type (REAL*8 or DOUBLE PRECISION) is written as an optionally signed string of decimal digits that includes an exponent and an optional decimal point. The exponent part consists of the letter D (or d) followed by an optionally signed integer constant. The integer represents the power of 10 by which the value is multiplied. The exponent must be included in the constant representation.

A double-precision real value occupies 8 bytes of storage. The precision is approximately 16 decimal digits. Both the range and precision of double-precision numbers are greater than those for real numbers.

The following are examples of the double-precision data type:

```
   +1.123456789d-2              1.D-2              1D-2
+Ø.ØØØØØØØØØØØØ1DØ    1ØØ.ØØØØØØØØ5D-3Ø5    .ØØØ12345D+123
```

The range of double-precision real values is:

Smallest Value	Largest Value	
2.23*10** – 308	1.79*10**308	Normalized
2.47*10** – 324	2.23*10** – 308	Denormalized

Complex

The *complex* data type (COMPLEX, COMPLEX*8, or COMPLEX*16) is written as an ordered pair of real, integer, or double-precision values. The values are enclosed in parentheses and can optionally be signed. An example of the complex data type is (*c1*,*c2*).

The first value in the pair (*c1* in the example) represents the real part of a complex number. The second value in the pair (*c2* in the example) represents the imaginary part of a complex number.

If either value in the pair is a double precision constant, the constant is typed as COMPLEX*16. A complex constant is stored as two consecutive real or double precision constants. For example:

(2.63,7.5) is equivalent to 2.63 + 7.5i

(-1Ø,-1E+3) is equivalent to -1Ø.Ø - 1ØØØ.Øi

(2,7.5DØ) is equivalent to 2.ØDØ + 7.5DØi

Note: The use of double precision values in a complex value is an extension to the ANSI X3.9-1978 FORTRAN 77 standard.

Logical

The *logical* data type is represented by the two logical values TRUE and FALSE. A logical data type is specified in IBM FORTRAN/2 as LOGICAL*1, LOGICAL*4, or LOGICAL. LOGICAL*1 specifies a 1-byte logical, while LOGICAL*4 and LOGICAL specify a 4-byte logical. There is no compiler option that sets a default for logical data storage allocation.

There are only two logical constants (.TRUE. and .FALSE.). They represent the two corresponding logical values.

The internal representation of a logical value uses 1 byte of storage. In LOGICAL*4 values, the least significant byte contains the value and the other bytes are undefined. If the logical value is .FALSE., the byte contains a hexadecimal 0. If the logical value is .TRUE., the byte contains the hexadecimal value FF. Hexadecimal values 01 through FE are normally treated as .TRUE., but they can cause errors and should be avoided.

Character

The *character* data type is written as a sequence of ASCII characters. The length of a character value is equal to the number of characters in the sequence. The length of a particular constant or variable is fixed, and must be between 1 and 32767 characters. A character variable occupies 1 byte of storage for each character in the sequence.

A *character constant* is one or more characters enclosed by a pair of apostrophes. Each character counts as one position. Blank characters are permitted in character constants and are significant.

An apostrophe within a character constant is represented by two consecutive apostrophes with no blanks between them. The length of a character constant is equal to the number of characters between the apostrophes, with doubled apostrophes counting as a single apostrophe character.

Examples of character constants are:

```
'A'
' '
'Help'
'can''t'
'A very long CHARACTER constant'
''''
```

In the last example, '''' represents a single apostrophe, '.

IBM FORTRAN/2 permits source lines of up to 72 columns. Shorter lines are padded to column 72 with blanks. Therefore, when a character constant extends across a line boundary, its value is assigned as if the line were padded with blanks to column 72 and placed before the continuation line.

For example,

```
200   CH = 'ABC
     1DEF'
```

is equivalent to

```
200   CH = 'ABC (57 blanks) DEF'
```

Note: IBM FORTRAN/2 also allows Hollerith and hexadecimal constants. See Appendix I, "Hollerith and Hexadecimal Data" in the *IBM FORTRAN/2 Language Reference* manual for information on using these types.

Symbolic Names

All symbolic names can have 1 to 31 characters. They must begin with an alphabetic character (including $) and can be followed by a mix of alphabetic, numeric, and underscore characters. Embedded blanks are ignored in a symbolic name, and no distinction is made between uppercase and lowercase letters.

Note: Symbolic names having more than 6 characters and the use of the dollar sign ($) and underscore (_) as a character in a symbolic name are extensions to the ANSI X3.9-1978 FORTRAN 77 standard.

IBM FORTRAN/2 keywords can be used as symbolic names and are identified by context. The concept of reserved words does not apply in IBM FORTRAN/2.

There are two main categories of symbolic names: *global* and *local*. Global names have meaning throughout an executable program. Local names are specific to a program unit.

Global Names

Global names identify global entities in one of the following classes:

- Common block
- External function
- Subroutine
- Main program
- Block data subprogram.

Local Names

Local names identify local entities in one of the following classes:

- Array
- Variable
- Constant
- Statement function
- Intrinsic function
- Dummy procedure.

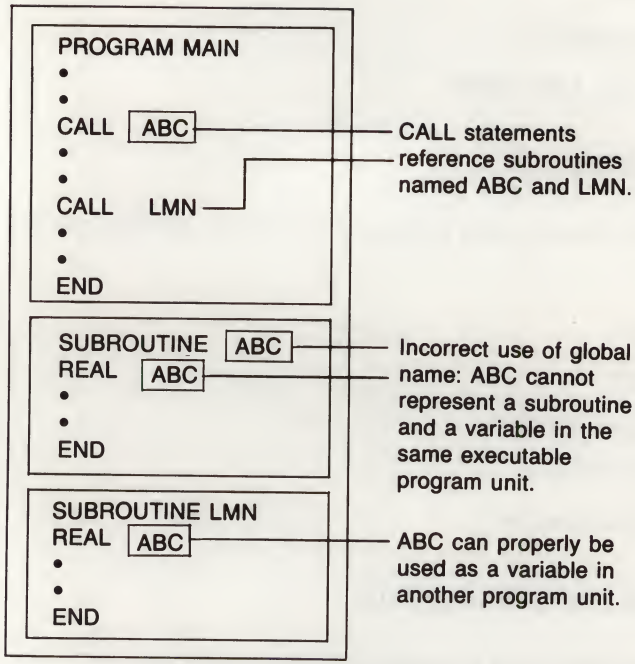
Symbolic Name Exceptions

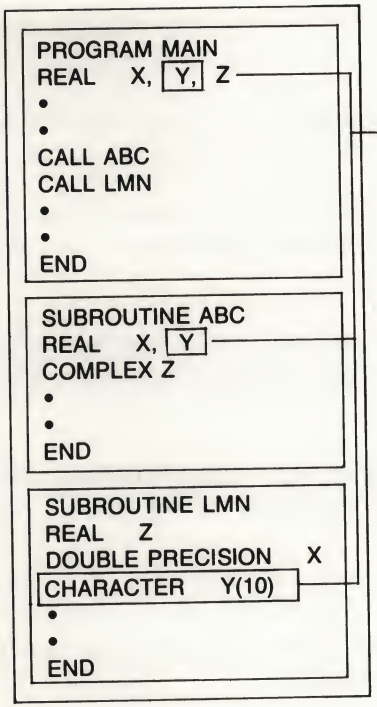
A symbolic name must be unique within its class, and can belong to only one class (global or local), except for the following:

- An external function name can be the same as a local variable name in the function.
- A common block name can also be an array, variable, dummy argument, or statement function name in the same program unit. A reference to the name in a COMMON or SAVE statement represents the common block name. Anywhere else, it is the local name.

A local name can be the same as a global name if the global name is not referenced within the program unit.

Examples


PROGRAM MAIN
CALL ABC
CALL LMN
END
SUBROUTINE ABC
REAL ABC
END
SUBROUTINE LMN
REAL ABC
END
CALL statements reference subroutines named ABC and LMN.
Incorrect use of global name: ABC cannot represent a subroutine and a variable in the same executable program unit.
ABC can properly be used as a variable in another program unit.


PROGRAM MAIN
REAL X, Y, Z
CALL ABC
CALL LMN
END
SUBROUTINE ABC
REAL X, Y
COMPLEX Z
END
SUBROUTINE LMN
REAL Z
DOUBLE PRECISION X
CHARACTER Y(10)
END
A local name can have a different meaning in each program unit.
Each is assigned separate storage.

Special Names

Names that have meaning for a single statement only are:

- Statement function dummy arguments
- DO-variables of implied-DO lists in a DATA statement.

A statement function dummy argument is used only within the scope of the statement function.

A DO-variable of an implied-DOlist appearing in a DATA statement has the scope of the implied-DO list.

A statement function dummy argument or DO-variable name can also appear as a local variable name or common block name in the same program unit.

The use of a name as a statement function dummy argument or DO-variable name removes the name from the class of intrinsic function names for the program unit.

The use of the IMPLICIT statement or of explicit type statements, including a length specification, applies to these names.

Establishing Data Types

A symbolic name has a definite data type in a program unit. The type can be any of those described previously:

- Integer [*2|*4]
- Real [*4|*8]
- Double precision
- Complex [*8|*16]
- Logical [*1|*4]
- Character.

Note: The types INTEGER*2, INTEGER*4, REAL*4, REAL*8, COMPLEX*8, COMPLEX*16, LOGICAL*1, and LOGICAL*4 are extensions to the ANSI X3.9-1978 FORTRAN 77 standard.

The symbolic name used to identify a constant, variable, array, or function (except for a generic function) also identifies its data type. A symbolic name can have only one type for each program unit. Once a type is established in a program unit, that type is implied for every reference to the same name.

The type of the name of a constant, variable, array, external function, or statement function can be established explicitly in an *explicit type statement* or by the IMPLICIT statement. The type of an external function name can also be established explicitly by the FUNCTION statement.

The data type of an array element name is the same as the data type of its array name. The data type of a function name determines the type of data that results from a function reference in an expression. For example, a reference to an integer function name results in integer data being returned to the point of reference.

The type of an intrinsic function name is given in Appendix D, "Intrinsic Functions" in the *IBM FORTRAN/2 Language Reference* manual. The name can also appear in an explicit type statement.

The IMPLICIT statement does not change the type of an intrinsic function. If the IMPLICIT NONE statement is used, all names (except intrinsic functions) must be assigned an explicit type.

Note: The IMPLICIT NONE statement is an extension to the ANSI X3.9-1978 FORTRAN 77 standard.

When the IMPLICIT NONE statement is not used and the type of a name is not established by the methods described above, the type is established *implicitly* according to the following rules:

- Symbolic names beginning with the letters I, J, K, L, M, or N are of type integer. For example:

```
J    I3    LAMBDA
```

- Symbolic names beginning with any other letters (including $) are of type real. For example:

```
X    Z1    ACCELERATION
```

Consequently, double-precision, complex, logical, and character type symbolic names must be assigned a type using explicit data statements or by the IMPLICIT statement. See the Type Statement and IMPLICIT Statement sections in Chapter 2, "Statements" in the *IBM FORTRAN/2 Language Reference* manual for more information.

Arrays

An *array* is a set of data having a name and a type. An array is defined with at least one and not more than seven dimensions. An array's name and dimensions are defined by an array declarator.

Array Names

The symbolic name of an array refers to the sequence of consecutive data called *array elements*. Each array element is referenced by the array name and a subscript. The value of the subscript specifies the particular element within the array. All the elements of an array are of the same type as the array name.

Array Declarator

An array declarator declares the symbolic name and the dimensions of an array. The two main types of array declarators are:

- Actual array declarator
- Dummy array declarator

They can be further classified as:

- Constant array declarator
- Assumed size array declarator
- Adjustable array declarator.

An *actual array declarator* can only be a constant array declarator. It is one in which the corresponding array is not a dummy argument in an external procedure. The actual array declarator is permitted in a DIMENSION statement, a TYPE statement, or a COMMON statement.

A *dummy array declarator* can be a constant array declarator, an assumed size array declarator, or an adjustable array declarator. It is one in which the array is a dummy argument in an external procedure.

The dummy array declarator is permitted in a DIMENSION statement and a TYPE statement, but not in a COMMON statement.

The format of an array declarator is:

a([*low1*:]*up1*[,[*low2*:]*up2*]...[,[*low7*:]*up7*])

where:

a is the symbolic name of the array.

lowi is an integer constant expression and defines the lower bound of dimension range *i*. (See "Arithmetic Expressions" on page 2-23 for more information.) When *lowi* is not specified, a value of 1 is assumed. The value of any *lowi* must be less than or equal to its corresponding *upi*.

upi is an integer constant expression and defines the upper bound of dimension range *i*.

The number of dimensions must be between one and seven, inclusive.

In an *assumed size array declarator*, the upper bound of the last dimension is an asterisk.

In an *adjustable array declarator*, the integer expression defining the upper or lower bound can include integer variables.

The following is an example of an array declarator that describes a three-dimensional array:

```
POLY (-5:5,6,2:11)
```

The first subscript expression in this example varies from −5 through 0 through 5, inclusively. The second subscript expression varies from 1 through 6, and the third subscript expression varies from 2 through 11.

Array Elements

The format for an array element name is:

a(*s1*[,*s2*]... [,*s7*])

where:

a	is the name of the array.
(*s1*[,*s2*]...)	is a subscript.
si	is a subscript expression.

A *subscript expression* is an integer expression used to identify a specific element of an array. The number of subscript expressions must match the number of dimensions in the array declarator.

Note: The value of a subscript expression must be between the lower and the upper bounds of the array. There may be unpredictable results if the subscript expressions exceed the specified bounds.

Examples

```
ARRAY(3,5)
X (I(N),J(N),K(N))
J (J(J(J(M))))
```

Array Dimensions

The number of dimensions in an array equals the number of dimension declarators in the array declarator.

The size of an array dimension is the value *up-low* + 1, where:

low	is the dimension's lower bound.
up	is the dimension's upper bound.

Array Element Storage

Elements of an array are stored consecutively in column-major order, so that the leftmost dimension varies most rapidly and the rightmost dimension varies least rapidly, even when the array has several dimensions.

The following table illustrates how array element names correspond to array positions:

	Array Declarator P(3,2,3)	**Array Declarator Q(0:2, – 2:3)**
Position	Element Name	Element Name
1	P(1,1,1)	Q(0, – 2)
2	P(2,1,1)	Q(1, – 2)
3	P(3,1,1)	Q(2, – 2)
4	P(1,2,1)	Q(0, – 1)
5	P(2,2,1)	Q(1, – 1)
6	P(3,2,1)	Q(2, – 1)
7	P(1,1,2)	Q(0,0)
8	P(2,1,2)	Q(1,0)
9	P(3,1,2)	Q(2,0)
10	P(1,2,2)	Q(0,1)
11	P(2,2,2)	Q(1,1)
12	P(3,2,2)	Q(2,1)
13	P(1,1,3)	Q(0,2)
14	P(2,1,3)	Q(1,2)
15	P(3,1,3)	Q(2,2)
16	P(1,2,3)	Q(0,3)
17	P(2,2,3)	Q(1,3)
18	P(3,2,3)	Q(2,3)

The subscript value corresponding to a given array element is equivalent to the position integer listed in the table above.

Examples

```
C ASSIGN THE 4TH ELEMENT OF ARRAY A
C (3 DIMENSIONAL) THE VALUE 3.8 AND
C THE 5TH ELEMENT THE VALUE 5.7
      A(4,1,1)=3.8
      A(5,1,1)=5.7
```

Array Size

The size of an array is the number of elements it contains. The number of elements is the product of the sizes of the dimensions specified by the array declarator. When an array is a dummy argument in an external procedure, it can be declared to be either a constant array, an assumed size array, or an adjustable array. See "Using Arrays as Arguments on page 3-10 for information about using assumed size and adjustable arrays as arguments.

Assumed Size Arrays

To find the number of elements in an assumed size array, use the actual argument (in the calling program unit) corresponding to the dummy array.

When the actual argument is a *noncharacter array name*, the size of the dummy array is the size of the actual argument array.

When the actual argument is a *noncharacter array element name,* the size of the dummy array is $x+1-r$, where:

- x is the size of the array containing the actual argument name.
- r is the subscript value of the actual argument.

When the actual argument is a *character array name, character array element name,* or *character array element substring name*, the size of the dummy array is INT$((c+1-t)/ln)$, where:

- c is the number of character storage units in the actual array.
- t is the character storage unit where the element or substring begins in the array.
- ln is the length of a dummy array element.

To check the calculated size of an assumed size array, set *n* equal to the number of dimensions in the array. The product of the sizes of the first *n-1* dimensions must be less than or equal to the size of the actual array.

The following initial lines of an external procedure illustrate the declaration of an assumed size array:

```
SUBROUTINE XRAY(A,B)
DIMENSION A(*),B(*)
•
•
```

One possible call to this subroutine is:

```
DIMENSION FX(20),SX(30)
•
•
CALL XRAY(FX,SX(10))
•
•
```

After this call, the assumed size arrays A and B have sizes 20 and 30 + 1 – 10 = 21, respectively. See "Using Arrays as Arguments" on page 3-10 for more information on associating dummy argument array elements and actual argument array elements.

Adjustable Arrays

An array that is in a subprogram routine and that has adjustable dimensions is an *adjustable array*. This is a dummy argument array that receives its dimension declarators as well as its name from the actual argument list of a calling program unit.

The following example of the transposition of a matrix depicts the use of adjustable dimensions.

Example

```
      SUBROUTINE TRNSPZ(X,Y,I,J)
      DIMENSION X(I,J),Y(J,I)
      DO 10 M=1,J
      DO 10 N=1,I
10    Y(N,M)=X(M,N)
      RETURN
      END
```

An appropriate call to this subroutine would be:

```
DIMENSION AMX(5,10),BMX(10,5)
•
•
CALL TRNSPZ(AMX,BMX,5,10)
•
```

When you want to use adjustable dimensions in a function or subroutine, you use a variable in the array declarator. Each dimension bound expression is formed from integer constants, symbolic names of constants, and variables. Any variable so used must appear in common or in all dummy argument lists in the subprogram.

When an array name or an array element is passed to a function or subroutine, the corresponding memory location is passed, not the data value. For this reason, dimension declarations between dummy and actual arrays need not be the same. In addition, an association can be established between a portion of an array in a calling program and a dummy array in a referenced program.

Using Array Names

An array name can be used alone, without subscripts, to identify properties of the entire array. You can use an array name in the following contexts:

- In COMMON, EQUIVALENCE, DATA, and SAVE statements
- In explicit type statements
- In an array declarator (though the declarator can have the same form as an element name, the declarator is not an array element name, by context)
- As a dummy argument
- As an actual argument
- In a data transfer statement's I/O list
- As a format identifier in a data transfer statement
- As a unit identifier for an internal file.

You cannot use an assumed size array in an I/O list, or as a format or unit identifier.

Character Substrings

A *character substring name* identifies a portion of a character variable or a character array element. The character positions in a string are consecutively numbered, in storage sequence order, beginning with position 1. (The character positions in a character constant are numbered from left to right.) A substring is a set of characters having consecutively numbered positions within a string.

A substring name has one of the following forms:

Character Variable	Character Array Element
v(*p1*:*p2*)	*a*(*s1*[,*s2*]...) (*p1*:*p2*)
v(*p1*:)	*a*(*s1*[,*s2*]...) (*p1*:)
v(:*p2*)	*a*(*s1*[,*s2*]...) (:*p2*)
v(:)	*a*(*s1*[,*s2*]...) (:)

where:

v	is a character variable name.
a(*s1*[,*s2*]...)	is a character array element name.
p1,*p2*	are *substring expressions*, that is, integer expressions that specify character positions.

The first substring expression, *p1*, designates the beginning character position of the substring. When *p1* is omitted, its value is 1. The second substring expression, *p2*, designates the ending character position of the substring. When *p2* is omitted, its value is the length of the string, *L*.

The values of *p1* and *p2* must be as follows:

$1 <= p1 <= p2 <= L$

The length of the character substring is: $p2 - p1 + 1$.

Note: IBM FORTRAN/2 does not always check the values of *p1* and *p2*. There may be unpredictable results if *p1* and *p2* exceed the specified range.

Examples

```
ARRAY(3,6) (:4)
ABC (3:4)
CHARID(I-1:I+1)
XQUOTE(K:)
WHOLE (:)
```

Each of these examples identifies a character substring.

The first substring is a 4-character substring in the character array named ARRAY. It consists of the first 4 characters of the array element named ARRAY(3,6).

The second substring is a 2-character substring starting at the third character position in character variable ABC.

The third is a 3-character substring starting at the character position immediately preceding character position I in character variable CHARID.

The fourth is a variable length substring starting at character position K and extending to the last character position in character variable XQUOTE.

The last substring contains all of the characters in character variable WHOLE.

Storage Allocation

When storage allocation instructions are not declared in a program, variables and arrays are allocated in the order: scalars, arrays, equivalence groups. The amount of storage allocated depends on the data type of the variable or array. The storage requirements of IBM FORTRAN/2 data types are listed below. If an item requires more than one storage unit, the allocated units are consecutive.

Type	Storage (Bytes)
LOGICAL	4
LOGICAL*1	1
LOGICAL*4	4
INTEGER	2 or 4 (depending on the /I compiler option)
INTEGER*2	2
INTEGER*4	4
CHARACTER	1
CHARACTER*n	n
REAL	4
REAL*4	4
REAL*8	8
DOUBLE PRECISION	8
COMPLEX	8
COMPLEX*8	8
COMPLEX*16	16

Note: The data types LOGICAL*1, LOGICAL*4, INTEGER*2, INTEGER*4, REAL*4, REAL*8, COMPLEX*8, and COMPLEX*16 are extensions to the ANSI X3.9-1978 FORTRAN 77 standard.

You can control how variables and arrays are stored by using the COMMON and EQUIVALENCE statements. The COMMON statement allows a block of storage to be shared among program units. The EQUIVALENCE statement associates more than one symbolic name with the same physical storage. See the COMMON statement and EQUIVALENCE statement sections of Chapter 2, "Statements" in the *IBM FORTRAN/2 Language Reference* manual for more information on storage allocation statements.

Static and Dynamic Storage

Two types of storage are assigned to variables and arrays within a program: *static storage* and *dynamic storage*. Static storage is assigned at the start of program execution and remains dedicated until the program stops. Dynamic storage is allocated when a block of executable statements is entered, and is released when control transfers from that block.

For dynamic storage, the data may not be the same as the last time the block was exited. For static storage, the data would remain the same.

For IBM FORTRAN/2, all data declared in a program unit is static. For other implementations of FORTRAN 77, the data associated with a subprogram or a named common block may be static or dynamic. In those implementations where the data may be dynamic, the SAVE command can be used to force the data to be static. See the SAVE statement section of Chapter 2, "Statements" in the *IBM FORTRAN/2 Language Reference* manual for more information about saving variables and arrays in subprograms.

Note: Since all storage in IBM FORTRAN/2 is treated as static, the SAVE statement is unnecessary. However, use of the SAVE statement is recommended for compatibility, if values are to be retained.

Expressions

An expression is formed from operands, operators, and parentheses. Operands are constants, variables, array elements, substrings, function references, and subexpressions. Subexpressions are expressions enclosed in parentheses. Operators are either *unary* or *binary*. Unary operators act on a single operand. Binary operators act on two operands.

The evaluation of an expression always yields a single result.

IBM FORTRAN/2 has four classes of expressions:

- Arithmetic
- Character
- Relational
- Logical.

Arithmetic Expressions

An arithmetic expression produces a value that is either integer, real, complex, or double-precision. The simplest forms of arithmetic expressions are:

- An unsigned integer, real, complex, or double-precision constant, or the symbolic name of such a constant
- An integer, real, complex, or double-precision variable
- An integer, real, complex, or double-precision array element
- An integer, real, complex, or double-precision function reference
- An arithmetic subexpression.

The value of a variable or array element must be defined if it is to appear in an arithmetic expression. In addition, the value of an integer variable must be defined with an arithmetic value, rather than a statement label value (set in an ASSIGN statement). See the ASSIGN statement section of Chapter 2, "Statements" in the *IBM FORTRAN/2 Language Reference* manual for more information.

Note: There may be unpredictable results if you use a statement label value as an arithmetic value.

Other arithmetic expressions are built up from the simple forms in the preceding description using parentheses and the arithmetic operators shown in the following table:

Arithmetic Operators

Operator	Operation	Precedence
**	Exponentiation	High
*	Multiplication	Intermediate
/	Division	Intermediate
+	Addition or Identity	Low
–	Subtraction or Negation	Low

All are binary operators, appearing between their arithmetic expression operands. The + and – can also be *unary,* preceding their operand.

Operations of equal precedence are evaluated from left to right except exponentiation, which is evaluated from right to left. For example:

```
A/B*C
```

is the same as:

```
(A/B)*C
```

and:

```
A**B**C
```

is the same as:

```
A**(B**C)
```

You can form arithmetic expressions in the usual mathematical sense, as in most programming languages. However, in IBM FORTRAN/2, two operators cannot appear consecutively. For example:

```
A**-B
```

is prohibited, although:

```
A**(-B)
```

is permissible.

Note that unary minus is also of lowest precedence, so that:

```
-A**B
```

is interpreted as:

```
-(A**B)
```

rather than as:

```
(-A)**B
```

An arithmetic constant expression is an arithmetic expression in which each numeric operand is either a numeric constant, the symbolic name of such a constant, or an arithmetic constant expression enclosed in parentheses. Exponentiation is allowed, but the exponent must be of type integer.

Use of Parentheses

The use of parentheses in an expression can control the order of evaluation of the expression. When part of an expression is enclosed in parentheses, that part is evaluated first, according to the order of precedence of the operators. If there is more than one level of parentheses, the innermost subexpression is evaluated first. With each result, that set of parentheses is removed. Subexpressions at the same level are evaluated from left to right. The resulting value is then used in the evaluation of the remainder of the expression.

Example

```
(((4+3)*2-10)+7)/(4-15)=
(( 7   *2-10)+7)/(4-15)=
((   14  -10)+7)/(4-15)=
(     4      +7)/(4-15)=
       11       /  -11 =-1
```

The use of parentheses to specify the evaluation order is often important in high-accuracy numerical computation. In such computations, evaluation orders that are algebraically equivalent might not be computationally equivalent when processed by a computer.

Type Conversions and Result Types

When all operands of an arithmetic expression are of the same type, the value produced by the expression is also of that type. When the operands are of different data types, the value produced by the expression is of a data type determined by a rank, as follows:

Data Type	Rank
COMPLEX*16	Highest
COMPLEX[*8]	
REAL*8 (DOUBLE PRECISION)	
REAL[*4]	
INTEGER*4	
INTEGER	
INTEGER*2	Lowest

Except for the combination of REAL*8 and COMPLEX*8, any operation that has two arithmetic operands of different data types has a resulting value which is the highest ranked data type in the operation. For example, an operation containing an integer and a real element produces a real result. When an operation has one operand of type REAL*8 and another operand of type COMPLEX*8, the result is of type COMPLEX*16. An expression using operands of different types is called a *mixed mode expression*. All combinations of numeric operands are valid.

Note: Mixed mode expressions containing complex and double-precision are an extension to the ANSI X3.9-1978 FORTRAN 77 standard.

Integer operations are performed on integer operands only. If the quotient of two integer operands is not an integer, the result is truncated towards zero rather than rounded.

Example

```
I=7
J=2
K=-7
L=I/J
M=K/J
```

When the expressions are evaluated, L = 3 and M = −3. Also, note that the expression:

```
1/4 + 1/4 + 1/4 + 1/4
```

has a value of 0.

Real operations are performed on real operands or combinations of real and integer operands. When an operation has a real and integer operand, the integer operand is first converted to a real data type by giving it a fractional part equal to zero. Real arithmetic is used to evaluate the expression. For example, in:

```
A  =  N/B
```

the value N is converted to a real data type and real division is performed on N and B.

When an expression contains mixed data types, the integer and real operations that are performed are evaluated in the order of precedence of the operators. For example, in:

```
Y = X * (I + J)
```

integer addition is performed on I and J, the sum is changed to a real data type, and real multiplication is performed on the result and X.

Note: IBM FORTRAN/2 does not check for integer overflow. You can get unpredictable results if you exceed the limits of integer values.

The data type of a subexpression is not affected by the data type of the context in which the subexpression appears. For example, if I is integer and R is real in the expression:

```
R+(I*2)
```

the subexpression I*2 is evaluated with an integer result before the addition is evaluated. When a subexpression contains several operations, each operation is performed according to the type of its two operands. The final data type of the subexpression is the highest rank in the subexpression.

Conversions to higher rank take place according to the following rules:

- In an integer to real conversion, the integer becomes the whole part of the real number, and the fractional part is set to zero.
- In a real to double-precision conversion, accuracy is not increased. Binary zeros are added in the low order positions to increase the length to 8 bytes, and the length of the exponent is adjusted at the same time.
- Any number, except COMPLEX*16, is converted to single precision complex (COMPLEX or COMPLEX*8) by first being converted to single-precision real. This result then becomes the real part of the complex number, and the imaginary part is set to zero.
- A single-precision complex number is converted to COMPLEX*16 by both the real and imaginary parts being converted to double precision. Any other number is converted to COMPLEX*16 by first being converted to double precision. The result becomes the real part of the complex number, and the imaginary part is set to zero.

Note: More precision is maintained when the computation occurs in the math coprocessor. Use the alternate functions SNGL and DSNGL to force truncation, if desired.

Character Expressions

A *character expression* produces a value that is a character data type. The simplest forms of character expressions are:

- Character constant
- Constant symbolic name
- Character variable
- Character array element
- Character substring
- Character function reference
- Character subexpression.

Other character expressions are built up from the simplest forms using parentheses and the character concatenation operator.

Character Concatenation

The character concatenation operator (//) joins two character data types. The format for using this operator is:

c1[//*c2*]...[//cn]

where:

cn is any form of a character expression.

Character expressions are evaluated from left to right. The result of a concatenation operation is a new string which appends *c2* to the right-hand end of *c1*. The length of the new string is the sum of the lengths of *c1* and *c2*. Parentheses that are not part of a string have no effect on the value of a character expression.

The character expression *c1* cannot have an asterisk (*) as its declared length except as follows:

- When used within a character assignment statement. (This can never occur in a main program unit.)
- When *c1* is the symbolic name of a character constant.

Examples

```
    Expression          Result     Length

'ABC'//'DEFGHI'       'ABCDEFGHI'     9
'AB'//('CD'//'EF')     'ABCDEF'       6
```

See "Data Types and Constants" on page 2-10 for more information on character data.

Character Constant Expression

A *character constant expression* is a character expression containing one or more of:

- A character constant
- A symbolic name of a constant
- A character constant expression enclosed in parentheses
- The concatenation of the above.

Relational Expressions

Relational expressions compare the values of two arithmetic or two character expressions. The result of a relational expression is a logical value that is either true or false.

Relational Operators

Relational expressions can use any of the following operators to compare values:

Operator	Representing Operation
.LT. or <	Less than
.LE. or <=	Less than or equal to
.EQ. or ==	Equal to
.NE. or <>	Not equal to
.GT. or >	Greater than
.GE. or >=	Greater than or equal to

Note: The use of <, <=, ==, <>, >, and >= as relational operators is an extension to the ANSI X3.9-1978 FORTRAN 77 standard.

All of the operators are binary operators, appearing between their operands. The periods are part of the operators and must be included. There is no relative precedence among the relational operators.

The following are examples of valid expressions. The value of each expression is either true or false.

```
A  .GT.  B

X  .EQ.  7
```

Arithmetic Operands in Expressions

Relational expressions can have arithmetic operands of different types. All combinations are valid.

Note: Mixed mode expressions containing complex and double precision are an extension to the ANSI X3.9-1978 FORTRAN 77 standard.

The operand of the lower type is converted to the operand of the higher type before the relational expression is evaluated. If either operand is of type complex, the relational operator must be either .EQ. or .NE. .

Character Relational Expression

Relational expressions with character operands compare the position of their operands in the ASCII collating sequence. An operand is less than another if it appears earlier in the collating sequence. When operands of unequal length are compared, the shorter operand is extended to the length of the longer operand by adding blanks.

Example

```
'AB ' .EQ. 'AB'
```

The result of the example is true.

The comparison is performed on a character-by-character, left-to-right basis until one of two situations occurs:

- All characters have been compared and found to be equal. In this case, the strings are considered to be equal (.EQ.)
- A pair of characters is found to be not equal. In this case, the strings are considered to be not equal (.NE.).

In the case of strings that are not equal, the string whose character has the lower hierarchy in the ASCII collating sequence is considered to be less than the other string. Thus, the collating sequence defines the results of relational expressions using character string operands.

Example

```
'NAME' .GT. 'N'
```

The result of this example is true.

Logical Expressions

A logical expression produces a value that is a logical data type. The simplest forms of logical expressions are:

- Logical constant or symbolic name of a logical constant
- Logical variable
- Logical array element
- Logical function reference
- Relational expression.

Logical Operators

To build up logical expressions from the preceding simple forms, use parentheses and the logical operators as follows:

Operator	Operation	Precedence
.NOT.	Negation	Highest
.AND.	Conjunction	High
.OR.	Inclusive disjunction	Intermediate
.EQV.	Equivalence	Low
.NEQV.	Nonequivalence	Low

Assume P and Q are logical expressions. Use the following table to see how each operator works:

	P = true	P = true	P = false	P = false
	Q = true	Q = false	Q = true	Q = false
.NOT.P	false	false	true	true
P.AND.Q	true	false	false	false
P.OR.Q	true	true	true	false
P.EQV.Q	true	false	false	true
P.NEQV.Q	false	true	true	false

The .AND. and .OR., .EQV. and .NEQV. operators are binary operators, appearing between their logical expression operands. The .NOT. operator is unary, preceding its operand. Operations of equal precedence are evaluated from left to right.

Examples

```
A .AND. B .AND. C
```

is equivalent to:

```
(A .AND. B) .AND. C
```

As an example of the precedence rules:

```
.NOT. A .OR. B .AND. C
```

is interpreted the same as:

```
(.NOT. A) .OR. (B .AND. C)
```

Two .NOT. operators cannot be adjacent, although:

```
A .AND. .NOT. B
```

is an example of an allowable expression with two adjacent operators.

Note: .NEQV. is equivalent to an *exclusive or*.

Hierarchy of Expressions and Operators

The following operators are in order of highest precedence from the top of the list. Items within a box are of the same precedence. Note that subexpressions of any type are evaluated before any other operation.

Arithmetic Expressions	* *	Exponentiation
	*	Multiplication
	/	Division
	+	Addition
	–	Subtraction
Character Expressions	//	Concatenation
Relational Expressions	.GT.	Greater than
	>	
	.GE.	Greater than or equal to
	>=	
	.LT.	Less than
	<	
	LE.	Less than or equal to
	<=	
	.EQ.	Equal to
	==	
	.NE.	Not equal to
	<>	
Logical Expressions	.NOT.	Negation
	.AND.	Conjunction
	.OR.	Inclusive Disjunction
	.EQV.	Equivalence
	.NEQV.	Nonequivalence

Executable and Nonexecutable Statements

The statements that can be included in an IBM FORTRAN/2 program are classified as *executable* or *nonexecutable*.

Executable statements specify actions and form an execution sequence in an executable program.

Nonexecutable statements specify characteristics, arrangement, and initial values of data; contain editing information; specify statement functions; classify program units; and specify entry points within subprograms. They are not part of the execution sequence. Nonexecutable statements can be labeled. However, the statement labels (other than for FORMAT statements) cannot be referenced in the program.

Executable Statements

Executable statements fall into the following classes:

- Assignment statements
- Control statements
- I/O statements.

Assignment Statements

Assignment statements are executable statements that define data values. The data assignment statements are:

- ASSIGN statement
- Assignment (computational).

Control Statements

Control statements are executable statements that can be used to control the order of execution in a program unit. The control statements are:

- CONTINUE statement
- DO statement

- ELSE statement
- ELSE IF statement
- END statement
- END IF statement
- GO TO (Unconditional, Computed and Assigned) statements
- IF (Arithmetic, Logical and Block) statements
- PAUSE statement
- STOP statement
- CALL statement
- RETURN statement.

I/O Statements

Executable I/O statements define files, control data transfer between internal storage and external or internal files, and control file position. The executable I/O statements are:

- BACKSPACE statement
- CLOSE statement
- ENDFILE statement
- INQUIRE statement
- OPEN statement
- PRINT statement
- READ statement
- REWIND statement
- UNLOCK statement
- WRITE statement.

Note: There is also a nonexecutable I/O statement, the FORMAT statement. Its use is discussed in "Data Format Statement" on page 2-49.

Nonexecutable Statements

Nonexecutable statements fall into the following classes:

- Specification statements
- Compiler directive statements
- Program, subprogram, and function statements
- Data format statement.

Specification Statements

Specification statements provide information about data to the compiler. They describe symbolic names, and define storage locations, data types, and program structure. The specification statements are:

- DIMENSION statement
- EQUIVALENCE statement
- COMMON statement
- INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, and CHARACTER type statements
- IMPLICIT statement
- PARAMETER statement
- EXTERNAL statement
- INTRINSIC statement
- SAVE statement
- DATA statement.

Compiler Directive Statements

Compiler directive statements control page ejects in the listing, and allow source statements from secondary files to be included in a program. The compiler directive statements are:

- EJECT statement
- INCLUDE Statement.

Program, Subprogram, and Function Statements

The nonexecutable statements in this class mark the beginning and entry points of the main program unit and all subprogram units, and define statement functions. The statements are:

- PROGRAM statement
- ENTRY statement
- FUNCTION statement
- SUBROUTINE statement
- BLOCK DATA statement
- Statement function statement.

Data Format Statement

IBM FORTRAN/2 includes the FORMAT statement, which is a nonexecutable I/O statement. It is used with formatted READ, WRITE, or PRINT statements to describe the external representation of data.

Ordering of Statements and Lines

Statements and lines within an IBM FORTRAN/2 program are ordered according to certain rules. These rules are as follows:

- Comment lines can appear anywhere in a program unit, including ahead of its first statement or after the final END statement.

  Note: The use of comment lines after the final END statement is an extension to the ANSI X3.9-1978 FORTRAN 77 standard.

- A PROGRAM statement, if present, must be the first statement of a main program. The first statement of a subprogram unit must be a FUNCTION, SUBROUTINE, or BLOCK DATA statement.
- A FORMAT statement can appear anywhere.
- An ENTRY statement can appear anywhere in a function subprogram or subroutine subprogram, except within the range of a DO-loop or an IF-block.

- All specification statements (other than DATA) must precede all DATA statements, statement function definition statements, and executable statements.
- All statement function statements must precede all executable statements.
- DATA statements can appear anywhere after specification statements.
- An IMPLICIT NONE statement cannot be preceded or followed by another IMPLICIT statement.

  Note: The Implicit None statement is an extension to the ANSI X3.9-1978 FORTRAN 77 standard.

- Any statement specifying the type of a constant's name must precede a PARAMETER statement defining the same constant name. A PARAMETER statement must precede all other statements referencing the symbolic names of constants that appear in it.

  Note: Allowing IMPLICIT statements to follow specification statements (other than PARAMETER statements) is an extension to the ANSI X3.9-1978 FORTRAN 77 standard. It is recommended that IMPLICIT statements precede all other specification statements for compatibility.

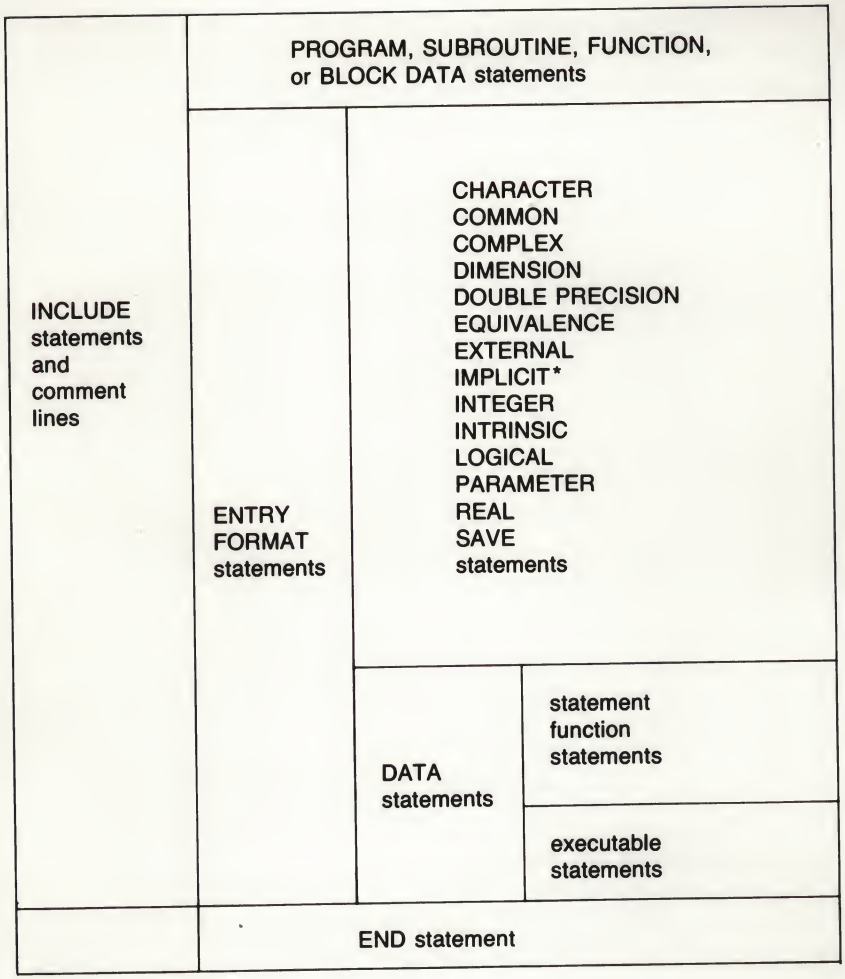
- The last line of any program unit must be an END statement.

Ordering Rules Illustrated

The rules for ordering statements are summarized in the following table. The table's vertical lines delineate statement types that can be freely intermixed, while horizontal lines delineate statement types which must not be intermixed.

For example, FORMAT statements can be interspersed with statement function statements and executable statements.

The table also shows that a statement function statement must not be interspersed with executable statements.



***Note:** Allowing IMPLICIT statements to follow specification statements (other than parameter statements) is an extension to the ANSI X3-9 1978 FORTRAN 77 standard. It is recommended that IMPLICIT statements precede all other specification statements for compatibility.

For complete descriptions of the statements that can be included in an IBM FORTRAN/2 program, and explanations of their coding requirements, see Chapter 2, "Statements" in the *IBM FORTRAN/2 Language Reference* manual.

Chapter 3. Program Structure

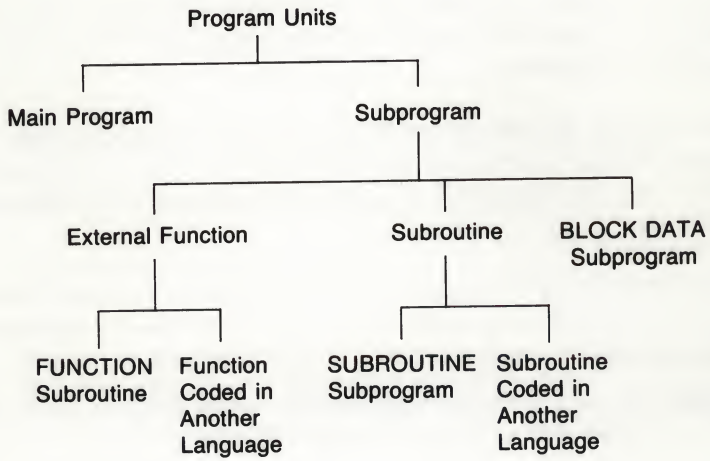
This chapter describes IBM FORTRAN/2 program structure. It explains:

- Program units
- Procedures
- Actual and dummy arguments.

Program Units

The complete set of IBM FORTRAN/2 code that is performed in a single program run is called an *executable program.*

An executable program consists of *program units.* A program unit is any number of executable or nonexecutable statements where the final statement is an END statement.



A program unit can be a *main program* or a *subprogram*. A main program begins with any statement except a FUNCTION, SUBROUTINE, or BLOCK DATA statement. A subprogram begins with a FUNCTION, SUBROUTINE, or BLOCK DATA statement, and is called a function, subroutine, or block data subprogram, respectively. The block data subprogram is nonexecutable. All other subprograms are executable.

An executable program is made up of:

- One main program
- Zero or more external procedures
- Zero or more block data subprograms.

Main Program

A *main program* is the program unit that receives program control from the processor when it is initially loaded at runtime. In turn, the main program controls the execution of subprograms and other procedures.

A main program is not an external function, external procedure, or block data subprogram.

A main program can begin with a PROGRAM statement. It can contain any set of statements except the FUNCTION, SUBROUTINE, BLOCK DATA, or ENTRY statements.

All variables and arrays defined in the main program are assigned to static storage, so the SAVE statement has no effect on their values. See "Storage Allocation" on page 2-31 for information about storage allocation.

Execution of a STOP statement, RETURN statement, or END statement in a main program halts the program run. (STOP and RETURN statements are optional in a main program. An END statement is required.)

Note: Allowing a RETURN statement in a main program is an extension to the ANSI X3.9-1978 FORTRAN 77 standard.

A main program cannot be referenced from itself or from a subprogram.

Subprograms

A subprogram is an independent program unit that is not a main program. External functions and subroutines are executable program units and can receive and return program control. A block data subprogram is a nonexecutable program unit. It provides initial values for variables and arrays in labeled common blocks.

Variables and arrays defined in subprograms are assigned to static storage, so the SAVE statement has no effect on their values. See "Storage Allocation" on page 2-31 for information about static and dynamic storage.

External Functions

An external function that is written in FORTRAN is a function subprogram. It begins with a FUNCTION statement and contains a series of executable statements. A function reference in an expression transfers control to a function subprogram. A RETURN or END statement returns control to the calling program unit. Execution of a STOP statement halts the program run.

A function returns a single value to the calling program unit by assigning that value to a variable having the same name as the function. The type of the function's name determines the data type of the value returned.

See the sections on the FUNCTION, ENTRY, and RETURN statements in Chapter 2, "Statements" in the *IBM FORTRAN/2 Language Reference* manual for information about defining and referencing external functions.

A function can be coded in assembly language. For more information, see "Rules for Coding Your Assembly Language Subprogram," in Chapter 8, and "Interfaces with Other IBM Languages and Products" in the *IBM FORTRAN/2 Compile, Link, and Run* manual.

Function Side Effects

A *side effect* is a consistent result of a function that is in addition to the basic result. A function side effect must not change the value of any other entity in the same statement. An actual argument or an associated argument defined during execution of a function cannot appear elsewhere in the referencing statement. The logical IF statement is an exception where a function reference in the logical expression can affect the contingent statement.

Subroutines

A subroutine that is written in FORTRAN is a subroutine subprogram. It begins with a SUBROUTINE statement and contains a series of executable statements. You use a CALL statement to transfer control to a subroutine, and a RETURN or END statement to transfer control to the calling program unit. Execution of a STOP statement halts the program run. A subroutine does not have a single data type or resulting value associated with its name.

See the sections on the SUBROUTINE, ENTRY, and RETURN statements in Chapter 2, "Statements" in the *IBM FORTRAN/2 Language Reference* manual for information about defining and referencing subroutines.

A subroutine can be coded in assembly language. For more information, see "Rules for Coding Your Assembly Language Subprogram," in Chapter 8, and "Interfaces with Other IBM Languages and Products" in the *IBM FORTRAN/2 Compile, Link, and Run* manual.

Block Data Subprograms

A block data subprogram is a program unit that begins with a BLOCK DATA statement and ends with an END statement. It contains a series of specification statements that define and initialize variables and arrays in labeled common blocks.

The block data subprogram differs from other subprograms because it does not contain any executable statements and neither receives nor returns program control. Because a block data subprogram is nonexecutable, it is not a procedure.

See the BLOCK DATA statement section of Chapter 2, "Statements" in the *IBM FORTRAN/2 Language Reference* manual for more information about block data subprograms.

Procedures

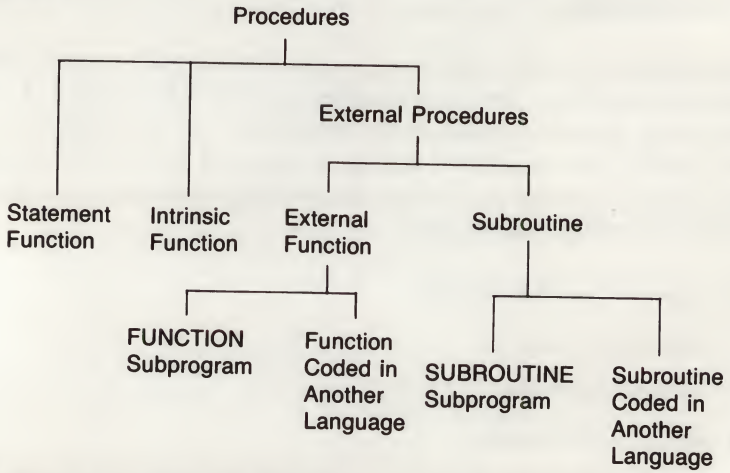
Procedures are entities that can be called by program units to perform particular activities. When a procedure name is referenced in a statement, control transfers to the procedure. The executable statements in the procedure are performed, and control returns to the reference. The program unit that contains the referencing statement is called the *calling program unit*.

The types of procedures are:

- Statement functions
- Intrinsic functions
- External procedures
  - External functions
  - Subroutines.

Statement functions, intrinsic functions, and external procedures are called *functions*. A function is referenced as a basic element in an expression. It acts upon zero or more quantities, called its *arguments*, to produce a single result, called the *function value*. The function value is returned to the calling program unit through the function name.

External functions and subroutines are *external procedures*. The following diagram shows the relationship between various types of procedures:



See Chapter 8, "Interfaces with Other IBM Languages and Products" in the *IBM FORTRAN/2 Compile, Link, and Run* manual for information about coding functions and subroutines in other languages. See the EXTERNAL statement section of Chapter 2, "Statements" in the *IBM FORTRAN/2 Language Reference* manual for information about OS EXTERNAL functions and subroutines.

Statement Functions

A *statement function* is a single-statement function that is internal to the program unit in which it is defined. A statement function reference returns a single value to its point of reference. Thus, a statement function can be included as a basic element in an expression.

The data type of a statement function's result matches the type of its name. The type of the statement function's name can be implicitly or explicitly declared, according to standard typing procedures. See the Statement Function Subprogram Statement section of Chapter 2, "Statements" in the *IBM FORTRAN/2 Language Reference* manual for information about defining and referencing statement functions.

Intrinsic Functions

Intrinsic functions are predefined functions supplied by the IBM FORTRAN/2 compiler. See Appendix D, "Intrinsic Functions" in the *IBM FORTRAN/2 Language Reference* manual for a complete list of the intrinsic functions and how to use them.

An actual argument of an intrinsic function is an expression. When an intrinsic function reference uses two or more actual arguments, good programming practice is to make all the arguments the same type. In most cases where they are not, the second argument is converted to the type of the first.

Note: This conversion is an extension to the ANSI X3.9-1978 FORTRAN 77 standard.

Every intrinsic function has an inherent type that is independent of implicit typing rules and depends only upon the function used. An IMPLICIT statement cannot change the type of an intrinsic function.

An intrinsic function can be referenced by a specific name or by a generic name. A *specific name* of a function requires a certain argument type and produces a result of a specified type. Except for a few intrinsic functions (for example, the type conversion functions), the type of the function's result agrees with the type of the arguments.

The result of a function referenced by a *generic name* depends on the type of the arguments. Not all intrinsic functions have generic names.

Usually, if the specific name of an intrinsic function appears in an INTRINSIC statement, the name can be used as an actual argument in an external procedure reference. (Exceptions are the names of intrinsic functions for type conversion, lexical relationship, bit manipulation, and choosing the largest or smallest value, and for argument passing.) An intrinsic function used in this manner is not classified as an external function. When a specific name is the same as the generic name, it is the specific function that is the actual argument, not the generic function.

External Procedures

An *external procedure* is an executable program unit that exists as an independent entity, and yet is not a main program. External procedures can be called by other elements of the executable program. The two types of executable procedures are external functions and subroutines. See "Program Units" on page 3-1 for a description of these procedures.

Executing an External Procedure

An external procedure is performed as follows:

1. The external procedure is called by a function reference or by a CALL statement, depending upon whether it is a function or subroutine.
2. Any expressions in actual arguments are evaluated.
3. The actual arguments from the calling statement pass to the dummy arguments of the external procedure. Thus, the dummy arguments are defined.
4. The statements in the program unit are performed in the order determined by the statements of the program unit.
5. Control returns to the calling program unit when either a RETURN statement or END statement is performed.

ENTRY statements provide secondary control transfer points in an external function or in a subroutine.

A function name can be used as a local variable in the body of the function. All local variable names that are the same as the function name or an entry point name are undefined when an external function begins to run. The referenced function name or entry point name must be defined during execution of the function. Once defined, the variable can be redefined. When control returns to the calling program, the value of this variable is returned as the function value.

The type of a function reference must agree with the type of the function name in the referenced subprogram. In the case of character functions, the length specifications must also agree. If the length specification of a local variable is an asterisk, a length conflict cannot

occur, since the length is determined by the length of the variable in the calling program unit.

A subroutine cannot be used as a basic element in an expression because it does not return a single value to the point of reference. Multiple results can be returned by functions as well as subroutines when dummy arguments become defined during subprogram execution. Functions and subroutines can return additional results to the calling program through entities in common storage.

Actual and Dummy Arguments

Values passed between calling program units and procedures are called *arguments*.

Arguments are either dummy arguments or actual arguments. A *dummy argument* is a symbolic name or an asterisk used in a procedure. Dummy arguments appear in a list where the procedure is defined (such as in a FUNCTION or SUBROUTINE statement). A corresponding *actual argument* appears in a list where the procedure is referenced. Dummy and actual arguments match in order, number and type.

No storage is reserved for dummy arguments. A dummy argument is only a name identifying the actual arguments passed to a procedure. When you call a procedure, the starting addresses of the actual arguments are substituted for dummy arguments before the procedure begins.

A dummy argument of an external function or subroutine can be:

- An array name
- A variable name
- A function or subroutine name
- An asterisk (subroutines only).

When a dummy argument is not declared to be an array or another procedure, it is treated as a variable.

Statement functions can only use variables as dummy arguments.

You can use a dummy argument name wherever you would use an actual name of the same type and class, except when explicitly prohibited.

Dummy argument names cannot be used in EQUIVALENCE, DATA, PARAMETER, SAVE, or INTRINSIC statements. In COMMON statements, they can be used only as common block names.

When you use character variables as arguments, the length of the actual argument must be greater than or equal to the declared length of the dummy argument. You can use an asterisk to specify the length of the dummy argument. (See the Type Statement section of Chapter 2, "Statements" in the *IBM FORTRAN/2 Language Reference* manual for more information about specifying lengths for dummy arguments.)

Using Arrays as Arguments

You can pass an array to an external function or subroutine. A dummy argument representing an array must be declared in a DIMENSION or explicit type statement within the subprogram where the dummy array is used. The corresponding actual argument can be an array or an array element.

The size of the actual array must be larger than or equal to the declared size of the dummy array. Since only the starting address of the actual array is passed to the dummy array, the dimensions can differ, and so can the length of the elements.

An asterisk appears as the upper bound of the last dimension in a dummy array declarator used to dimension an assumed size array.

Integer dummy arguments can appear as dimension bounds in a dummy array declarator used to dimension an adjustable array.

When an actual argument is a *noncharacter array name*, the size of the dummy argument array must not exceed the size of the actual argument array, and each actual argument array element becomes associated with the dummy argument array element of the same subscript.

When an actual argument is a *noncharacter array element name* with a subscript value of *as*, the dummy argument array element with a subscript value of *ds* becomes associated with the actual argument array element that has a subscript value of *as* + *ds* – 1. See "Array Element Storage" on page 2-24 for a definition of *subscript value*.

The following associations result from the example under "Assumed Size Arrays" on page 2-26.

```
A(1)=FX(1),...,A(20)=FX(20)
B(1)=SX(10),B(2)=SX(11),...,B(21)=SX(30)
```

When an actual argument is a character array name, character array element name, or character array element substring name and begins at a character storage unit *acu* of an array, character storage unit *dcu* of an associated dummy argument array becomes associated with character storage unit *acu*+*dcu* –1 of the actual array argument.

See "Assumed Size Arrays" on page 2-26 and "Adjustable Arrays" on page 2-27 for more information about assumed size and adjustable arrays.

Using Procedures as Arguments

When you use a procedure name as a dummy argument, you can pass an actual procedure name through more than one level of procedure call.

When an actual argument is an intrinsic function, it must be identified in an INTRINSIC statement in the calling program. You must use the specific name of the intrinsic function, if it has one.

A generic name may not be used unless it is the same as a specific name (in which case, it is treated as the specific name). Refer to Appendix D, "Intrinsic Functions" in the *IBM FORTRAN/2 Language Reference* manual for more information about intrinsic functions.

When an actual argument is an external procedure name, it must be identified in an EXTERNAL statement in the calling program.

When a dummy argument corresponds to an actual function, the dummy function name must agree with the type of the actual function name.

Using Alternate Return Specifiers

An asterisk dummy argument indicating an alternate return can only appear in a SUBROUTINE statement or in an ENTRY statement within a subroutine.

The argument corresponding to an asterisk dummy argument must be an asterisk followed by the statement label of the alternate return point.

Chapter 4. File Processing

This chapter describes the IBM FORTRAN/2 file system. It explains the concepts of:

- Records and units
- Files and file access
- Elements of I/O statements
- Format considerations
- Device identifications.

This chapter is a guide for the use of the input/output (I/O) statements presented in Chapter 2, "Statements" in the *IBM FORTRAN/2 Language Reference* manual.

Records

The IBM FORTRAN/2 language defines three types of records:

- Formatted
- Unformatted
- Endfile.

Formatted

Formatted records are sequences of characters ended by the carriage return and linefeed. They are processed only by formatted I/O statements. Only formatted records can be used to pass data to or from the screen, keyboard, or printer.

The length of formatted records is measured in characters. Characters are stored in internal (binary) form. When formatted records are written, the items are converted from internal form to characters.

Unformatted

Unformatted records are sequences of values with no system alteration or interpretation; no physical representation exists for the end of the record. Unformatted records are used when you want to store or retrieve information without the need for editing or intervention. Only unformatted I/O statements can be used to process unformatted records.

Unformatted records are stored in internal (binary) form. Their length is determined by the number of values in the I/O list.

Endfile

An *endfile* record is a logical record, has no length, and is the physical and logical end of file. An endfile record is written by an ENDFILE statement. When it is read, the endfile record signals the end-of-file condition at runtime.

Files

A *file* is a logical collection of records. Files are either external or internal.

- An *external file* is a logical set of external physical records.
- An *internal file* is a character variable, character array, character array element, or substring that serves as the source or destination of some I/O action. It provides a method of converting formats within main memory. An internal file is always formatted and sequential.

  When the record is a substring or character variable, the file contains one record. When an internal file is a character array, each array element is a record in the file.

File Attributes

A file has these attributes:

- Name
- Structure (formatted or unformatted)
- Access method (sequential or direct)
- Position
- Shared
- Action.

Name

A file can have a name. If present, a name is a character string that is identical to the name by which the file is known to DOS or OS/2. See your operating system manual for more information on filename structures.

In addition to the IBM Personal Computer DOS and OS/2 filenames, the following special filenames exist:

- CON (display, keyboard I/O)
- AUX, COM1, COM2 (RS-232 I/O)
- PRN, LPT1, LPT2, LPT3 (printer).

IBM FORTRAN/2 also supplies preconnected filenames when you do not assign a filename yourself.

See "Device Identifications" on page 4-31 for more information about standard device connections and preconnected filenames.

Structure

An external file is opened as either formatted or unformatted. All internal files are formatted.

- A *formatted file* consists entirely offormatted records
- An *unformatted file* consists entirely of unformatted records.

Access Method

The access method determines the order in which records are read or written in a file. Access often depends on the storage device used and the way records are organized within the file. In all cases, the access method refers to the manner in which the FORTRAN system addresses the file. This may or may not relate to how the hardware device addresses the file.

An external file can be opened as sequential or direct. Some files can have more than one allowed access method; others are restricted to only one access method. Both sequential and direct access methods can be used for disk files, but sequential access is required for internal files.

The file access method is established when an external file is connected to a unit. See "Units" on page 4-18 and the OPEN statement section of Chapter 2, "Statements" in the *IBM FORTRAN/2 Language Reference* manual for information about establishing file/unit connections. The access method remains in effect throughout the connection. Only one access method can be used for a file during one connection.

Sequential Files

Sequential files contain records arranged in the order in which they were written. The last record written is the last record in the file. Sequential files contain no logical gaps.

DOS and OS/2 attempt to extend sequential files if a record is written beyond the old terminating file boundary. This is successful if space remains on the physical device.

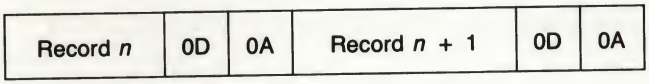
Only sequential files can contain an endfile record.

Records of a sequential file can vary in length.

Sequential files can be either formatted or unformatted.

Formatted Sequential Files

As shown below, each formatted sequential record written to the disk ends with the carriage return/line feed characters (ASCII characters 0D and 0A respectively). Since these characters are interpreted as an end-of-record marker, they should never appear within the record.



Carriage Control Characters

When you request output to the standard output device (unit 6 or *), the first character of each record is replaced by the carriage control characters appropriate for that device. There are no trailing end-of-record markers. You must provide a carriage control character from the following list:

Character	Effect
Blank	Advances one line
0	Advances two lines
1	Advances to top of next page
+	Carriage return only (allows overprinting)
–	None (suppress carriage control)

Note: A carriage control character of – (minus) is an extension to the ANSI X3.9-1978 FORTRAN 77 standard.

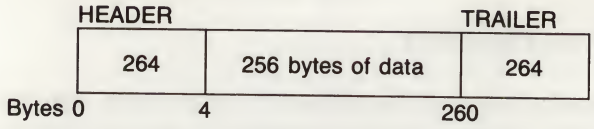
Any other character is treated as a blank, and advances one line.

Note: If unit 6 is opened as a file other than system standard output, the carriage control characters are placed at the front of the record. The first character from the record is then lost. Because of the file's different structure, it may not be suitable for further processing. However, it may be printed or displayed.

Unformatted Sequential Files

Each record contains a 4-byte header and a 4-byte trailer. The number of bytes in the record, including the header and trailer, is written into these 4-byte fields, called count fields.

For example, if the size of the actual record is 256 bytes, the length of the physical record is 264 bytes to account for the header and trailer. As shown below, the header and trailer each contain the value 264.



Direct-Access Files

Direct-access files are random-access files and can be read or written in any order. Each record has a *record number*, which identifies its logical position in the file.

The record number is an integer value that is specified when the record is written. Records are numbered sequentially. The first record is number 1.

All the records of a direct-access file must have the same length.

Any record in a direct-access file can be initialized or redefined in any order. No record can be deleted. The values of records that have not been written are undefined and should not be read.

If a direct-access file can also be connected for sequential access, it can contain an endfile record. The endfile condition is not recognized while the file is connected for direct access.

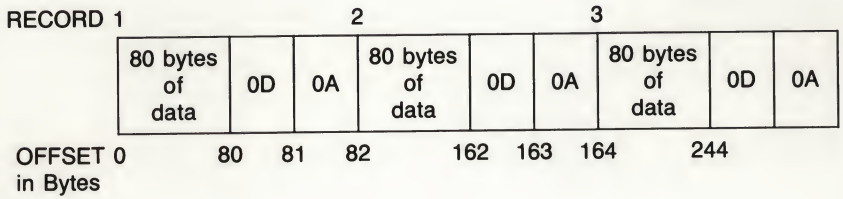
Note: There is no physical endfile record. If the file is read sequentially, FORTRAN/2 considers a logical endfile record to occur after the last physical record.

List-directed input/output cannot be used with direct-access files.

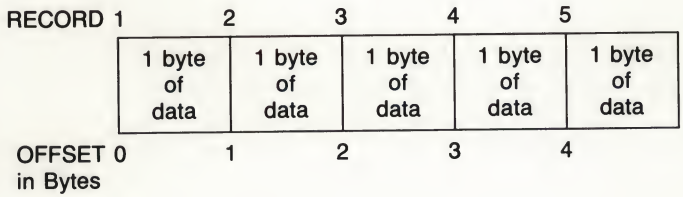
Direct-access files can either be formatted or unformatted.

Formatted Direct-Access Files

All records must have the same length, defined by the RECL specifier in the OPEN statement. (See the OPEN Statement section of Chapter 2, "Statements" in the *IBM FORTRAN/2 Language Reference* manual.) Characters in the record not explicitly written to are blank-filled by the runtime routines. Records longer than 1 byte end with trailing linefeed and carriage return characters (ASCII codes 0D and 0A, respectively). Therefore, the record length is the value of RECL+2. As shown below, the record corresponding to a record number (REC) is found at offset (REC-1) * (RECL+2) in the file.



Direct-access files with record lengths of 1 are an exception. If RECL=1, no carriage return and linefeed characters are added to the record. Therefore, each record length is 1. As shown below, the record corresponding to a record number (REC) is found at offset REC-1 in the file.

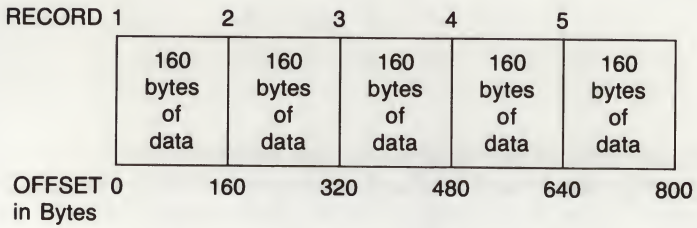


Unformatted Direct-Access Files

All records must have the same length, defined by the RECL specifier in the OPEN statement. (See the OPEN Statement section of Chapter 2, "Statements" in the *IBM FORTRAN/2 Language Reference* manual.) If less data is written than needed to fill the record, the remaining part of the record is undefined. No count fields are added to the records.

Note: Since there are no count fields in unformatted direct-access records, you must provide them if you plan to process the file sequentially. If you create an unformatted file using the sequential access method, you must consider the count fields to be data fields if the file is opened for direct access.

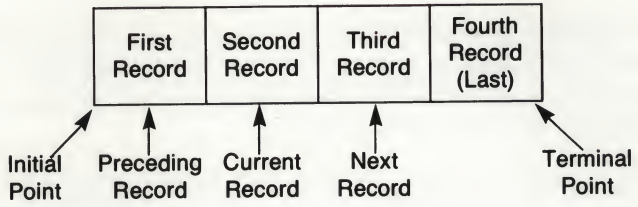
As shown below, the record corresponding to a record number (REC) is found at offset (REC – 1) * RECL in the file. In the example, RECL = 160.



Position

A file that is connected to a unit has a position property. (See "Connection" on page 4-17 for information about how a connection is established.) The position of a file is usually set by a previous I/O activity. As shown below, a file has an initial point, terminal point, current record, preceding record, and next record. The position of a file can also become indeterminate.

File Position



The *initial point* is the position before the first record. The *terminal point* is the position after the last record.

The *current record* exists when the file is positioned within a record. Otherwise, there is no current record.

The *preceding record* is the record before the current file position. The preceding record does not exist when the file is positioned at its initial point or in the first record of the file.

The *next record* is the record after the current file position. The next record does not exist when the file is positioned at the terminal point or in the last record of the file.

File Position before Data Transfer

Before data is read from a sequential file, the file is positioned at the beginning of the next record. This record becomes the current record when you issue a read. When data is written to a sequential file, a new record is created and becomes the last record of the file.

Data transfer cannot be performed on a file positioned after an endfile record.

An internal file is always positioned at the beginning of the first record of the file.

Before data is transferred to or from a direct-access file, the file is positioned at the beginning of the record specified by the REC specifier.

File Position after Data Transfer

If no error condition or end-of-file condition exists, the file is positioned after the last record read or written. That record becomes the preceding record. A record written on a file connected for sequential access becomes the last record of the file.

If reading an endfile record causes an end-of-file condition, the file is positioned after the endfile record.

Data transfer statements cannot be executed when the file is positioned after the endfile record. However, a BACKSPACE or REWIND statement can be used to reposition the file.

The file position is undefined when an error condition exists.

Shared Files

This section introduces the concept of shared files as it applies to IBM FORTRAN/2 programs. Details on opening a file for shared access are found in the OPEN statement section of Chapter 2, "Statements" in the *IBM FORTRAN/2 Language Reference* manual.

Note: Shared files are an extension to the ANSI X3.9-1978 FORTRAN 77 standard.

Record Locking

In order to allow file sharing, a set of record locking procedures must be put in place to ensure the integrity of data written by the programs sharing the file. Record locking allows a program to read and write a record without concern that, in between the read and write, another program reads or writes that record.

Generally, a record is locked when a program reads the record, and stays locked until another I/O operation (such as a write or a close) occurs. If an ERR or IOSTAT specifier is present in the READ statement, an error (RECORD LOCKED) occurs for the statement. If neither an IOSTAT nor an ERR specifier is provided and the record is locked, the program will wait. However, if after 9 seconds the lock is not removed, an error (RECORD LOCKED FOR 9 SECONDS) occurs. For an explanation of the IOSTAT and ERR specifiers, see the READ statement section of Chapter 2, "Statements" in the *IBM FORTRAN/2 Language Reference* manual.

An airline reservations systems provides a good example of the problems unlocked records can cause. Each record in the system represents an airline seat. Without record locking, two programs could read the same record and discover the same seat empty. Program One reserves the seat for a passenger. Program Two then comes along and, having found the seat empty during its read, also reserves the seat for its passenger. This overwrites the reservation placed in the record by Program One, now leaving its passenger without a reservation.

Locking the record between the read and write by the first program to enter the file prevents this problem. Subsequent programs are unable to read that record before the first program has a chance to write it.

Establishing Shared Files

As explained in the OPEN statement section of Chapter 2, "Statements" in the *IBM FORTRAN/2 Language Reference* manual, shared files are established through three interrelated specifiers. They define the file to be:

1. Shared (the SHARE specifier)
2. Opened for read, write or read/write (the ACTION specifier)
3. Opened for sequential or direct access (the ACCESS specifier).

Specifying SHARE='YES' does not guarantee that the file can be shared. For instance, if a program opens a file without the SHARE specifier or with SHARE='NO', the file will not be shared even if another program tries to open it with SHARE='YES'. Similarly, if a program opens a file for sharing with the write action and sequential access, another program will not be allowed to open the file for sharing and direct access. The interrelationship between specifiers is highlighted below, and further explained under the OPEN statement in Chapter 2, "Statements" of the *IBM FORTRAN/2 Language Reference* manual.

Shared Files for Sequential Access

A file opened for sequential access may be shared between programs if all the programs sharing the file request a read value for their ACTION specifiers.

Note: The other programs may open the file for sequential or direct access if both access methods are allowed for the file.

A program which requests a WRITE or READ/WRITE value for the ACTION specifier will not be allowed to open a file for sequential access, unless no other programs are currently connected to the file. That is, if WRITE or READ/WRITE is requested, a YES value for the SHARE specifier is ignored, and a value of NO is used.

Since shared sequential files may only be read, record locking is not done for files opened for sequential access.

Shared Files for Direct Access

A file opened for shared, direct access may be shared between programs under one of the following conditions:

1. All the programs sharing the file are opened for direct access (ACCESS = 'DIRECT').
2. All the programs sharing the file are opened with the read action (ACTION = 'READ').

When a file is opened for shared direct access with read/write action (ACTION = 'READ/WRITE'), IBM FORTRAN/2 employs the following technique to ensure data integrity:

1. On each read of the file:
   a. The record in the file that was last locked but not yet unlocked by the program is unlocked.
   b. If the record to be read is not currently locked, the record is locked and read.
   c. If the record to be read is currently locked by another program and:
      1) If neither an IOSTAT nor an ERR specifier exists, the program waits approximately 9 seconds for the record to be unlocked, then locks and reads the record. If the record cannot be locked after 9 seconds, an error occurs and execution terminates.
      2) If an IOSTAT or an ERR specifier exists, the read fails with an error.

2. On each write to the file:

    a. If the record to be written by a program is the same as the last record locked but not yet unlocked by that program, the write is performed and the record is then unlocked.

    b. If the record to be written by a program is not the same as the last record locked by that program:

        1) The record in that file that was last locked but not yet unlocked by the program is unlocked.

        2) If the record to be written is not currently locked, the record is written.

        3) If the record to be written is currently locked by another program and:

            a) If neither an IOSTAT nor an ERR specifier exists, the program waits approximately 9 seconds for the record to be unlocked, then writes the record. If the record is not unlocked after 9 seconds, an error occurs and execution terminates.

            b) If an IOSTAT or an ERR specifier exists, the write fails with an error.

3. If the file is closed (by either an OPEN or CLOSE statement), or an UNLOCK statement is performed for the file, the record in the file that was locked but not yet unlocked by the program is unlocked.

4. If an error occurs during an I/O operation (other than INQUIRE) on the file, the record in the file that was locked but not yet unlocked by the program is unlocked.

5. If the program terminates, the record in the file last locked but not yet unlocked by the program is unlocked.

A program will not lock any record in a file opened for shared direct access and with read or write action. Locks placed on the file by other programs will be checked on reads and writes in the following manner:

1. If the record is not locked, it will be read or written immediately.
2. If the record is locked by another program and:
   a. If neither an IOSTAT nor an ERR specifier exists, the program waits approximately 9 seconds for the record to be unlocked, then reads or writes the record. If the record is not unlocked after 9 seconds, an error occurs and execution terminates.
   b. If an IOSTAT or an ERR specifier exists, the read or write fails with an error.

Avoiding Deadlock

For a single shared file, locks provide an automatic way to control access to the file. For multiple shared files, the programs must be coded to avoid problems that can arise. The following demonstrates the problem commonly called "deadlock" or the "deadly embrace".

Two programs do the following:

Program A

1. Locks record 1 in file S (by reading the record)
2. Writes to record 2 in file T

Program B

1. Locks record 2 in file T (by reading the record)
2. Writes to record 1 in file S

A deadlock will occur if the above is executed in the following order:

1. Program A locks record 1 in file S.
2. Program B locks record 2 in file T.
3. Program A attempts to write record 2 in file T (but waits for B to remove the lock).

4. Program B attempts to write record 1 in file S (but waits for A to remove the lock).

The programs are now waiting for each other. To prevent this problem, follow these rules:

1. Unlock records as soon as possible. This can be done implicitly by executing another I/O operation to the file, or explicitly by executing an UNLOCK statement for the file.
2. If the program has a lock on a record in another file, then before writing to a record in another shared file, lock the record by reading it first.
3. If more than one shared file will have a locked record at the same time, always lock records in the files in the same order for all programs which share the files.

For example, the above programs could be changed to:

Program A

1. Locks record 1 in file S (by reading the record)
2. Locks record 2 in file T (by reading the record)
3. Writes to record 2 in file T

Program B

1. Locks record 1 in file S (by reading the record)
2. Locks record 2 in file T (by reading the record)
3. Writes to record 1 in file S

In this case, deadlock cannot occur.

Deadlock can also be prevented by having IOSTAT or ERR specifiers (or both) in the I/O statements. However, appropriate corrective action must be taken. (If the action is just to retry the I/O, deadlock may still occur.)

The following example shows deadlock occurring when the third rule is not followed:

Program A

1. Locks record 1 in file S (by reading the record)
2. Locks record 2 in file T (by reading the record)

Program B

1. Locks record 2 in file T (by reading the record)
2. Locks record 1 in file S (by reading the record)

If the steps are executed in the following order, a deadlock will occur:

1. Program A locks record 1 in file S.
2. Program B locks record 2 in file T.
3. Program A attempts to lock record 2 in file T but waits until Program B's lock is removed.
4. Program B attempts to lock record 1 in file S but waits until Program A's lock is removed.

Again, this can be prevented by having programs lock records in shared files in the same order.

Elements of I/O Statements

The three types of I/O statements are:

- Data transfer statements
- Auxiliary I/O statements
- File positioning statements.

I/O activities affect, and are affected by, the existence and connection status of a file.

Units

A *unit* is simply a way of referring to a file. IBM FORTRAN/2 refers to files by the unit numbers specified in I/O statements. An external unit identifier is a non-negative integer expression or an asterisk. An internal unit specifier is the name of a character substring, variable, array, or array element.

File Existence

An external file *exists* when it is assigned to an external device. An internal file always exists.

An external file can exist but contain no records if none have been written yet.

Creating a file causes it to exist when it did not previously. *Deleting a file* ends its existence.

A file can exist and yet not be connected to a unit.

All I/O statements can specify files that exist. OPEN, CLOSE, INQUIRE, WRITE, PRINT, REWIND, UNLOCK, and ENDFILE statements can also specify files that do not exist. File creation can occur as a result.

Connection

The association of a unit with a data file is called a *connection*. A unit can be connected to only one file at a time. A file can be connected to only one unit at a time. You can connect a unit and a file explicitly or implicitly.

The OPEN statement makes an explicit unit/file connection. For example, the following statement connects unit number 8 to the file named OLD.F10:

```
OPEN (UNIT=8, FILE='OLD.F10')
```

The OPEN statement can connect a unit to a system device using a reserved name. The following OPEN statement connects unit 9 to the printer:

```
OPEN (UNIT=9, FILE='LPT1')
```

Note: OS/2 reserved names must be used without a colon.

An asterisk used as a unit specifier indicates a system standard I/O device. The keyboard (CON) is the system standard input device. The display (CON) is the system standard output device. Unit numbers 5 and 6 are also implicitly connected to the system standard input and output devices, respectively, if they are not explicitly connected to other files. Note that unit number 5 and "*", when used for input, are identical. (The explicit connection of unit 5 to a file also causes subsequent input from "*" to be from the same file.) The same relationship holds between unit 6 and "*", when used for output.

When an OPEN statement does not specify a filename, or if a unit is not explicitly opened, IBM FORTRAN/2 makes an implicit unit/file connection, using a preconnected filename. Implicit file/unit connections are established before the program run.

See "Device Identifications" on page 4-31 for more information about standard device connections and preconnected filenames.

All I/O statements except OPEN, CLOSE, and INQUIRE must specify units that are connected to a file.

Disconnection

Units and files are *disconnected* when the unit/file connection ends. You can disconnect a unit and a file explicitly or implicitly.

A unit and file are explicitly disconnected by a CLOSE statement specifying the unit.

A unit and file are implicitly disconnected by an OPEN statement specifying the same unit with a different filename. Any connected files are also implicitly disconnected at the end of a program run.

Data Transfer Statements

Data transfer statements transfer data between a file and internal storage. Transferring data from an internal or external file to storage is called *reading*. Transferring data from storage to an external or internal file is called *writing*. Reading is an input operation, and writing is an output operation.

The data transfer statements are READ, WRITE, and PRINT. They are performed as follows:

1. Establish the direction of the data transfer.
2. Determine the unit.
3. Set up the format reference, if one is specified.
4. Set the file position before the data transfer.
5. If the file is subject to locking, remove the old lock and set a new lock, if required. (See "Record Locking" on page 4-10.)
6. Perform the data transfer between the file and the I/O list items, if any.
7. Set the file position after the data transfer.
8. Set the value of the IOSTAT specifier, if one is specified.

Values are transferred between data records and I/O list items during the data transfer. Each list item is processed completely before the next is processed. Items are processed in the order they appear in the list. Therefore, an item defined by an input statement can be used in defining another item appearing later in the input list.

Observe the following restrictions for data transfer:

- An input list item cannot define part of a format specification used in the same statement.
- An I/O list item of an internal file cannot be contained within the file.
- An output list item must be defined before the output statement is performed.
- All input list items are undefined when you attempt to read a direct-access record that has not been defined.
- A format specification that is within an internal file cannot be specified in an output statement to the same internal file.

Control Information List

The *control information list* is used in a READ, WRITE, or PRINT statement to specify the following information about the I/O operation:

- Unit
- Format

- Record number
- I/O status
- Error return
- End-of-file return.

The control information list is a group of keywords and variables providing information about each of the attributes named above. The keywords and variables are called *specifiers*. Specifiers are described in detail under each I/O statement in Chapter 2, "Statements" in the *IBM FORTRAN/2 Language Reference* manual.

Input/Output List

An input/output list (I/O list) is used in a READ, WRITE, or PRINT statement to identify the data to be transferred. See the sections on these statements in Chapter 2,"Statements" in the *IBM FORTRAN/2 Language Reference* manual for descriptions of I/O list items.

Unformatted Data Transfer

An unformatted data transfer statement does not have a format specifier in its control information list.

An unformatted data transfer statement reads or writes one record. Data is transferred without editing. Only external files can be specified in unformatted I/O statements.

There must be at least as many values in the record as there are in the input list during unformatted input. Excess data in the record is ignored, but insufficient data causes an error. The values in the record must agree with the types of the input list items.

The size of an unformatted, sequential output record is determined by the size of the output list.

The size of an unformatted, direct-access output record is predetermined. The output list must not contain more items than the record can hold. If the output list contains too few values, the record remains partially undefined without error.

Formatted Data Transfer

A formatted data transfer statement has a format specifier in its control information list. See "Format Specifications" on page 4-23 and Chapter 3, "I/O Editing" in the *IBM FORTRAN/2 Language Reference* manual for information on format specifications and edit descriptors.

A formatted data transfer statement reads or writes at least one record. Data values are edited under format control before being stored or after being retrieved. Internal and external files can be specified in formatted I/O statements.

A formatted input operation reads a data record just before editing it. The input list determines the amount of data read. The corresponding format specification determines the position and form of the data. Excess data in the record is ignored, but insufficient data may cause an error. Trailing blanks are supplied for formatted sequential input. The format specification can cause a single formatted input statement to read more than one record. The record number of a direct-access file increases by one for each record read.

Note: Supplying trailing blanks for formatted sequential input is an extension to the ANSI X3.9-1978 FORTRAN 77 standard.

A formatted output operation edits one or more data records before transfer to an output file. The output list and corresponding format specification determine the amount of data written. A data value is retrieved from storage, converted to external form according to its corresponding repeatable edit descriptor, and placed in the output record. The output operation is complete when all output list items are converted and a repeatable edit descriptor or colon is reached in the format specification. Unused repeatable edit descriptors are ignored. The completed record is transferred to an external file or moved to an internal file. The format specification can cause a single formatted output statement to write more than one record.

Unreferenced character positions in a direct-access or internal file are filled with blanks. Output editing must not generate more characters than the record can hold. The record number of a direct-access file increases by one for each record written.

Auxiliary I/O Statements

Auxiliary I/O statements explicitly open or close a file, inquire about the status of a file or unit, or remove a lock from a shared file. The auxiliary I/O statements are OPEN, CLOSE, INQUIRE, and UNLOCK. The specifiers used in auxiliary I/O statements are described in detail in Chapter 2, "Statements" in the *IBM FORTRAN/2 Language Reference* manual.

File Positioning Statements

The file positioning statements manipulate file position for external sequential files. The files can be formatted or unformatted, but system standard files cannot be specified.

The file positioning statements are BACKSPACE, ENDFILE, and REWIND. The specifiers used in file positioning statements are described in detail in Chapter 2, "Statements" in the *IBM FORTRAN/2 Language Reference* manual.

Format Specifications

Formatted data transfer statements use *format specifications* to direct the editing between internal data representation and character strings of records in a file. A format identifier requests formatting in one of two ways:

- A format identifier that is not an asterisk requests *explicit formatting* according to a format specification within the same executable program unit.
- An asterisk requests implicit, *list-directed formatting*.

Explicit Formatting

A format specification is established by a FORMAT statement or by a set of character data forming a format specification.

A format specification has the form:

([*t*[[,]*t*]...])

where:

t is a repeatable edit descriptor, a non-repeatable edit descriptor, or a group format specification preceded by an optional repeat count.

A formatted input or output record is described in terms of fields. Each field is an individual data item. A record is composed of zero or more fields. The *field width* is the size of the field in characters.

On input, field width typically specifies the number of characters to be read from the input record. In IBM FORTRAN/2, numeric and logical fields may be terminated by a comma. In this case the field is interpreted as having leading blanks added. These blanks serve to lengthen the field to the specified width. The field width cannot be overridden on output.

Note: Field termination with a comma is an extension to the ANSI X3.9-1978 FORTRAN 77 standard.

Example

```
      READ (*,99) I,J
   99 FORMAT (2 I 1Ø)
```

Input record:

Position	01	02	03	04	05	06	07	08
Value			1	2	,	1	6	,

After read: I = 12 and J = 16.

Each field is edited by a repeatable edit descriptor, apostrophe edit descriptor, or H edit descriptor. Non-repeatable edit descriptors (except the apostrophe edit descriptor and H edit descriptor) are not associated with data fields. They determine such characteristics as carriage control, the end of the record, and the significance of blanks.

A data transfer statement can reference a FORMAT statement with its statement label or with an integer variable assigned the statement label value. See Chapter 2, "Statements" in the *IBM FORTRAN/2 Language Reference* manual for information on the FORMAT statement.

Format Stored as Character Data

You can create a format specification at runtime by storing it as character data. If the specification is filled by an input operation, the input must be performed before the character format specification is referenced. A character format specification can also be initialized by the DATA statement or by using character assignment statements.

A format specification can be stored in any of the following:

- A character array
- A character variable
- A character array element
- A character expression
- A numeric array that is initialized with a READ or Hollerith DATA statement.

The character data must have the same form as a format specification. Therefore, it must be a set of characters designating repeatable edit descriptors and their separators, and other edit descriptors. The data must be enclosed in parentheses, which can be preceded by blanks and followed by any characters.

If a variable or array element is used, the format specification must be completely contained within the item. If an entire array is used, the format specification can continue beyond the first element into following consecutive elements. A character array format specification is a concatenation of the array elements in storage sequence.

List-Directed Formatting

Using *list-directed formatting*, you can read and display information without using an explicit format specification. List-directed I/O uses a default formatting scheme to read or write data. An asterisk format specifier indicates list-directed formatting.

List-Directed Input

As in any formatted READ statement, input list items in a list-directed READ statement are defined by corresponding values in the record. Input values can be numeric, logical, and character constants, and null values.

List-directed input values are separated by commas, blanks, and slashes. Commas and slashes can be preceded and followed by blanks.

Numeric Values. An integer is represented as a string of decimal digits. A plus or minus sign is optional.

A real or double-precision value is represented by a field suitable for F editing. See Chapter 3, "I/O Editing" in the *IBM FORTRAN/2 Language Reference* manual for information about F editing. The value is assumed to have no fractional digits unless the input field contains an explicit decimal point.

Complex constants in list-directed input records are two constants (integer, real, or double precision) enclosed in parentheses and separated by a comma. Blanks can appear before and after the real and imaginary parts. This is the only time you can embed blanks in a numeric value in a list-directed input record. In any other case, digits separated by a blank are treated as two distinct values. The end of the record can also separate the two parts of the complex value, occurring either before or after the comma.

Examples

–12345	integer, real, or double-precision
–12345.	real or double-precision
45D–6	real or double-precision
(1.5,2)	complex

Logical Values. Any sequence of characters beginning with a T or F represents a logical value. A decimal point in front of the letter is optional.

Logical values cannot contain any blanks, commas, or slashes in a list-directed input record.

```
FAL/.SE
.T  RUE.
.TR,UE.
```

are invalid logical values.

Character Values. A character value is represented just as any character constant. It can contain blanks, commas, and slashes within the identifying apostrophes. An apostrophe must be represented by two consecutive apostrophes.

When the character constant in the record is longer or shorter than the declared length of the input list item, the constant is truncated or blank-padded on the right.

Null Values. A null is identified with two successive commas.

Examples

```
T, 47.6,12,,Y,,428
```

A null value assigned to an input element has no effect. The input value is kept if it is defined, or remains undefined if it never had a value.

Repeated Input Values. You can use a repeat count indicator when the same value repeats consecutively in a list-directed input record. Use the following form:

*r**[*c*]

where:

r is an unsigned, nonzero integer constant and describes a repeat count.

c is a constant value.

For example, 3*123 declares that the integer 123 repeats three times in the input record. When no constant is specified, a null value is used. 15* describes 15 nulls.

Special Characters. Most values in a list-directed input record are separated by a comma and any number of blanks.

Examples

```
1, -23  T, .F,  45.6
```

A slash in a list-directed input record marks the end of the input list and assigns nulls to any items remaining in the list.

Unless a comma or slash identifies a null input value, leading blanks in the first record are ignored in a list-directed data transfer.

When values do not fit in one input record, they can be continued on successive records, but only a complex or character value can be split.

When the end of a record is not within a character constant, it is treated as a blank.

A character value can be continued in as many records as required. The end of a record does not cause insertion of a blank or any other character.

Example

```
'YOU CAN CONTINUE A VERY LONG CHARACTER CONST
ANT THROUGH SEVERAL RECORDS      WITHOUT HAVI
NG ANY EFFECT ON ITS VALUE.
```

If each line represents a record, the constant in the example uses four records. The words split by the ends of the records still have the value 'CONSTANT' and 'HAVING' in internal storage. The last record is completely blank, but it still has value because it is within the apostrophes defining the constant.
The end of a record cannot separate two apostrophes representing a literal apostrophe.

Example

```
123, 456, 789.984, 'A B C   ' 93248.123 'DON'
'T SPLIT A CONSTANT AT DOUBLE APOSTROPHES'
```

There is no limit to the number of records that can be used to contain one constant.

List-Directed Output

List-directed WRITE and PRINT statements produce the values in the order they are named in the output list. The type of an output list item determines its written form.

Since the purpose of list-directed formatting is quick visual display, output records are edited assuming that carriage control is required. Thus, the first character of every output record is a blank to indicate single line spacing. Except for character values, one blank separates each value within the record.

Numeric Values. Integers are written using I11 as an edit descriptor. See Chapter 3, "I/O Editing" in the *IBM FORTRAN/2 Language Reference* manual for information about edit descriptors.

Real list items use an edit descriptor of 0PF16.8 when the magnitude of the value is $\geq 10^{**}(-1)$ and $<10^{**}6$. When the magnitude of a real value is outside that range, the field is edited to 1PE16.8.

Double precision values use an edit descriptor of 0PF25.17 when the magnitude is $\geq 10^{**}(-1)$ and $<10^{**}6$. When it is outside that range, the scale factor is set to one (1PE25.17).

COMPLEX or COMPLEX*8 items are written as two real values enclosed in parentheses and separated by a comma. COMPLEX*16 items are written as two double precision values enclosed in parentheses and separated by a comma. No blanks (other than those produced by the real or double precision editing formats) are included anywhere within the parentheses. A complex value is never split between records unless it is longer than an entire record.

If the complex value is longer than an entire record, the value is split after the comma. As in any list-directed output record, one blank begins the new record.

Logical Values. List-directed output records write a T or F for each logical value. The default edit descriptor is L1.

Character Values. Output character values are always written in their entirety when you use list-directed formatting. Character data uses A as an edit descriptor.

Because list-directed output fields are edited for display, character constants are written without enclosing apostrophes. An apostrophe within the constant is displayed as a single symbol.

Character values neither begin nor end with a value separator. Thus, a character constant is "printable" but not "readable". If you write a list-directed record to a disk, you must add apostrophes, spaces, and double apostrophes to any character constants before you can read the record with a list-directed READ statement. You can, of course, read it with an appropriate formatted READ.

Example

```
PRINT *, '''','DON''T',''''
```

The example PRINT statement produces the following record:

```
'DON'T'
```

When a character field is shorter than an entire record, the field is not split between records, but begins a new record when necessary.

When a single character value is longer than a record, the value continues from one record to the next. The blank character for carriage control is still written at the beginning of each continuation record.

When there are not enough character positions in the output record to hold the next value, the value begins the next record.

List-directed output does not use nulls, and a slash is written only when it is part of a character constant. Slashes do not separate values in a list-directed output record.

Device Identifications

An asterisk used as a unit specifier indicates the system standard input or standard output device. The keyboard is the system standard input device, and the display is the system standard output device.

Implicit file/unit connections use preconnected filenames, depending on the unit number specified in the I/O statement causing the connection and on the specific I/O statement used.

The unit numbers and the associated filenames are:

Unit number	Preconnected filename
0	DOS and OS/2 standard error output device CON when the I/O statement is a WRITE FORT0 when the I/O statement is a READ
1 - 4	FORT1 - FORT4

5	DOS and OS/2 standard input device (CON) when the I/O statement is a READ or FORT5 when the I/O statement is a WRITE.
6	DOS and OS/2 standard output device (CON) when the I/O statement is a WRITE or FORT6 when the I/O statement is a READ.
7 - 3275	FORT7 - FORT3275

IBM FORTRAN/2 also allows you to specify the asterisk as a unit number default:

- On input as the standard input device
- On output as the standard output device.

DOS and OS/2 limit the quantity of files that may be open at one time. For more information about the number of external files that can be open at one time, see "I/O Restrictions" in Appendix G, "Limits and Ranges" in the *IBM FORTRAN/2 Language Reference* manual.

Certain cautions and regulations must be observed when using default filenames.

- Unit 0 is the DOS and OS/2 standard error output device. However, it can be connected to a different filename if you so indicate in your OPEN statement. For example:

  ```
  OPEN (UNIT=Ø)
  ```

  connects unit 0 to the standard error file while the statement:

  ```
  OPEN (UNIT=Ø, FILE='NEWFILE')
  ```

  connects unit 0 to the file named NEWFILE.

  The statement:

  ```
  WRITE (UNIT=Ø)
  ```

  now writes to the file named NEWFILE, not to the standard error file.

- Reading from unit 0 creates a connection to FORT0 (not to standard error output).

- Unit numbers 5 and 6 default to the standard input and output files, respectively. However, specifying a filename in an OPEN statement allows you to override the default. For instance:

```
OPEN (UNIT=5)
READ (UNIT=5)
```

or

```
OPEN (UNIT=6)
WRITE (UNIT=6)
```

connect units 5 and 6 to the standard input and output files, respectively. (Note that this connection is made even without an OPEN statement.)

However, the statements:

```
OPEN (UNIT=5, FILE='NEWFILE')
OPEN (UNIT=6, FILE='NEWFILE1')
```

connect units 5 and 6 to the files named NEWFILE and NEWFILE1.

The statements:

```
READ (UNIT=5)
WRITE (UNIT=6)
```

now read from NEWFILE and write to NEWFILE1, not standard input and output.

Note: In this case, you cannot use * for either standard input or standard output. Overriding the default connection for unit numbers 5 and 6 overrides the connection for the asterisk as well.

- Writing to unit 5 or reading from unit 6 creates a connection to FORT5 and FORT6, respectively (not to standard input and standard output). It invalidates the use of * for standard input and standard output respectively.
- Using 'UNIT=*' creates a connection to standard input or standard output, depending on your FORTRAN statement.

Examples

```
READ (UNIT=*)
```

reads from standard input.

```
WRITE (UNIT=*)
```

writes to standard output.

Again, note that * cannot be used for standard input or standard output if the defaults for units 5 and 6 have been overridden.

Index

A

B

C

D

E

S

IBM United Kingdom
International Products Limited
PO Box 41, North Harbour
Portsmouth, PO6 3AU
England

Printed in Denmark by Interprint A/S